USENIX UNIX TRANSACTION PROCESSING WORKSHOP SPRING 1989

U
US
USE
U NIX
USENIX

WORKSHOP PROCEEDINGS

UNIX TRANSACTION PROCESSING

May 1 - 2, 1989
Pittsburgh, PA

For additional copies of these proceedings write

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710 USA

The price is $12

Outside the U.S.A and Canada, please add
$8 per copy for postage (via air printed matter).

Program and Table of Contents

UNIX Transaction Processing Workshop

Pittsburgh, PA

May 1-2, 1989

Sunday, April 30

Workshop Chair:

Douglas Kevorkian
AT&T Bell Laboratories
Room 5-340
190 River Road
Summit, NJ 07901

(201) 522-5086 (voice)
(201) 522-6621 (FAX)
attunix!dek

USENIX Workshop Coordinator:
Judith H. DesHarnais

USENIX Workshop Liaison:
Sharon Murrel

A Transaction Processing Monitor

Dan Hydar, David Scott, Felix Yang,[1] Morris Yeh
Marine Terminal Computer Systems
50 Beale Street
P.O. Box 3965
San Francisco, CA. 94119
E-mail: mtcs!tpm@uunet.uu.net

Introduction

This paper describes a transaction processing monitor that is the basis of a transaction processing system used for marine terminal applications. This system was developed to run under BSD-family UNIX. The applications supported by the system include distributed database and machine control. The "event"-based model offered by this system is described, as well as the underlying architecture that implements it.

In this paper, the word *system* will refer to the transaction processing system. The term *application* is used to denote a user-level program that is known to the system.

1 The Basic Model

Transactions

The most basic unit of work managed by the system is the *transaction* — an atomic grouping of operations performed by one process. A transaction is just a procedure that is given some number (possibly zero) of input parameters, does some sequence of operations and returns an *outcome* that consists of:

- A *result* of COMMIT if it was successful or ABORT if it was not.
- A *return code.* If a transaction results in ABORT, the return code will carry some information about why the transaction aborted. If the result is COMMIT, the return code can be used to flag warnings.
- Output data. If the result is COMMIT, the transaction is also allowed to return an arbitrary amount of data. This data is typically records retrieved from a database query or some information describing the result of the event in some way.

Operations done by transactions are usually database operations but are frequently device control operations as well. Typical examples of devices being controlled would be display screens or machines that move cargo from one place to another.

Applications that execute transactions are referred to as *server* processes; those that request transactions from other processes are called *requestor* processes. In general, a server manages a set of objects such as databases or machines and implements all transactions that operate on these objects.

[1] currently at Transmission Development Division, Fujitsu America, Inc.

Any application running in the system can, at any time, request a transaction or offer to execute one. Thus, the terms "server" and "requestor" should always be interpreted as being relative to a particular transaction.

Events

A requestor does not request a transaction directly, it requests a more general entity known as an *event*. An event is an atomic collection of one or more transactions. The transactions in an event can all execute at once and will either all commit or all abort. The transactions in an event are called its *related transactions*. The bindings between events and their related transactions are kept in an internal event table. Events are requested using a unique *event code*.

In order to guarantee atomicity of events, a two-phase commit protocol is used. Once the system finds one server per related transaction, the first (*prepare*) phase begins: the system sends each one a transaction request. Each server determines if the part of the event that its transaction performs can be done correctly. If so, the server is expected to set up an intermediate state from which either the original state or the new state can be obtained even if the system crashes. It then sends a COMMIT outcome of the transaction to the system; otherwise, it sends an ABORT. The second (*complete*) phase then begins. If all of the other transactions in the event also COMMITed, the event commits and the system will send a FINALCOMMIT message, telling the server to make the transaction permanent. Otherwise, the event aborts and the system will send a FINALABORT message telling the server to discard any effects of the transaction. In either case, the second phase message is sent to the requestor at the same time as the servers.

When an event commits its return code and output data buffer is that of one of its one transactions known as the *selected transaction*. Otherwise, the return code is that of the first transaction in the event to abort.

The event mechanism allows atomically-executing entities (events) to be made up of independent entities (transactions) and this buys a fair amount of flexibility, especially for incremental modification of events. For example, if the programmer wanted to add some functionality to an event e_1, she could do so by writing a new transaction, tx_i, that performed the new actions and then add tx_i to the definition of e_1 in the event table. The alternative to this would be to write an entirely new transaction that contained the code that implements the functionality of e_1 plus the code that would have been in tx_i. The advantage of the event-based solution is that it doesn't require any rewriting of (presumably) working code.

Passing Data Between Requestors and Servers

Information is passed between requestors and servers through an object known as a *data buffer*. A data buffer holds a set of bindings between symbol names and values. When an event is requested, a data buffer that contains the input parameters expected by the events transactions is passed. Each transaction that accesses the parameters passed by the requestor through the symbol name. A transaction's output data also takes the form of a data buffer.

Like the event mechanism itself, this method of passing parameters provides flexibility. If it is determined that one of the transactions included in an event requires more information, a new symbol can be added to the message and be passed with the data buffer. Since each transaction explicitly asks for the data by symbol name, other transactions in the same event are not affected by this change.

2 Operations

The interface offered to the application programmer by the transaction processing system consists of the following procedure calls. Whenever an application invokes one of these operations, it is sent a response message by the system.

- A requestor can *CALL* an event by specifying its event code and providing the input parameters needed for the event to execute. When the call is made, the system forwards it to a servers that implement its related transactions. The system response message contains the unique *event ID* that denotes the running event or an error code if the request failed.
- A requestor can *WAIT* on the result of a particular event, specified by its event ID. That is, the requestor can tell the system that it wants to be sent the outcome of the event. The system response is a message that contains the outcome of the event.
- A server tells the system that it is ready to receive a new transaction by using the *GET* operation. The system response is a copy of a request for an event that includes one of the servers transactions.
- The *END* operation is used by the server to return the outcome of a transaction it is executing. The system response is the *complete* message of the two-phase commit discussed above.

A *GET_RESPONSE* function is provided to be used by applications to obtain any system responses intended for them. Calling this function blocks the process until a message is available.

3 Transaction Processing Monitor

The backbone of this system is process known as the Transaction Processing Monitor (TPM). All event and transaction control is centralized in the TPM — any operations requested by an application involves exchanging messages with the TPM. The TPM provides several features.

- **Interprocess Communication:**

  The only inter-application IPC method supported by the system is the event mechanism. Thus, in this architecture, all IPC must go through the TPM. If one process wants to send a message to another, it actually does so by sending it to the TPM which forwards it to the other process.

  Communication is done using standard UNIX pipes. All IPC done by the TPM passes through these pipes. There is one input pipe for each application that is used to carry messages from the TPM to that applications. There is one TPM input pipe that is used by all applications to send messages to the TPM.

- **Event and Process Management:**

  There is a disk database that the TPM reads at startup time that describes the system. This database contains:

  - The definitions of all known events. That is, a mapping between event code and the list of transactions that are included in it.
  - A table of that describes the server programs that implement the transactions.

– A list of applications that are to be run at startup time.

When an event is requested, the requestor sends a message to the TPM containing the event's event code plus message buffer containing the input parameters for the event. The TPM then inspects its internal tables and broadcasts the message it received from the requestor to all of the servers that implement transactions that are part of the event.

At any point in time, the TPM is aware of which applications are running in the system, which are ready to perform a transaction for a requestor and which are currently executing transactions. If there are no running servers that can provide a requested transaction, the TPM uses the information in its database to start one. It is possible that some servers needed for a requested event are busy woking on some other event. If this is the case, the request is queued until the server is free.

- **Error recovery:**

  The TPM has a built in logging mechanism that keeps track of the status of non-completed event requests. Before sending the *CALL* response to the requestor, the TPM creates a record for the new event in its log file. This file contains information on all events and transactions that are run in the system. State information on the events run by the system and their related transactions is updated at significant points in their lifetimes (i.e. end of a phase in the two-phase commit protocol, etc.) The log file allows the TPM to recover events if there is a system crash.

4 Process Structure

To function properly, the TPM has to keep track of when processes terminate. Since a process may terminate accidentally due to program error, it is not possible to rely on the application process itself to inform the TPM that it is exiting. Special processes called *watchdogs* were created for this purpose. When an application is started, it is not forked by the TPM, but by the watchdog process. So that the pipe can be established, the watchdog must be forked by the TPM. Once the application process is forked, the watchdog sleeps. Since the watchdog is the parent process of the application process, it will receive a SIGCHLD signal if the application exits. When this occurs, the watchdog sends a message to the TPM's input pipe telling it that the application has terminated.

UNIX imposes a fixed limit on the number of file descriptors that a process can obtain. Since each link to an application requires one file descriptor, this would be a severe limitation on the number of processes in the system. In order to avoid this limitation, special processes were added to the system to provide extra communication paths. These processes, known as *Transaction Message Handlers* (TMH) are connected to the TPM with a pipe. A TMH is connected to some number application input pipes (one pipe per application). Each application is connected to exactly one TMH. When the TPM sends a message to an application, it does so by writing it to the TMH input pipe, the TMH then reads the message and writes it to the input pipe of the actual application. Using this scheme, if each TMH has P available pipes, the number of connections available for applications is increased by a factor of P.

There are some situations where it is not possible for an application to be started as a descendant process of the TPM, a prime example being remote processes. In order to support these, another class of system processes exists: the Special Message Handler (SMH). The SMH exists as a descendant of the TPM and communicates with it in the same way as discussed for applications. However, each SMH has a socket that is used to connect with applications that can't be descendants of TPM. The SMH acts as a two-way forwarder — whatever it reads from the socket, it forwards to the TPM; whatever it reads from the TPM, it writes to the socket.

5 Reducing IPC Overhead

The number of messages in the protocol caused a high overhead. To reduce this, the following additions were made to the system.

- A mechanism that allows a server participating in an event to tell the TPM that is does not need to participate in the second phase of the commit protocol. Since many of the transactions in a typical event don't have any work to in the second phase of the commit process, this saves quite a few messages.
- The protocol was modified to combine some frequently-associated operations to reduce the number of messages. For example, a requestor very frequently sends a *WAIT* for the result of an event immediately after it is called so a CALL_AND_WAIT option was added to the *CALL* operation to allow this to be done by passing only one message to the TPM. It is also typical for a server process to *GET* a transaction, process it, *END*, and then repeat. So an END_AND_GET option was added to the *END* operation. Using this option tells the system that the server can process a new transaction once the current one completes. Since the majority of applications are of the form described above, these "combined" operations significantly increase efficiency.
- A special subroutine interface, the Transaction Front End (TFE) was developed. The TFE allows server code to be compiled directly into the TPM without any modification. This is accomplished by offering two separate libraries : the *external* TFE and the *internal* TFE. The external version of the TFE is used by servers that are implemented as separate processes and need to communicate with the TPM. The internal TFE is linked in with the TPM code and provides the same functional interface to applications as the external, but all message passing is done with function calls and local buffers instead if IPC since the server code is actually present within the TPM process.

6 Things that would have helped

There are two things that were not available for the TPM that would have been useful in implementing an efficient system:

- **Broadcast** Our system frequently does operations that involve broadcasting identical messages to multiple processes. For example, the second phase of an event involves the TPM sending exactly the same message to every server participating in the event. All message-based IPC in UNIX is, unfortunately, point-to-point, which forces the TPM to do one write() per participating server. It could have been much more efficient if some sort of "write to multiple file descriptor" call was available.
- **Shared Memory** The availability of shared memory would have added the capability to distribute the work of the TPM across the processes in the system and reduce the cost of the IPC. If the TPM database could be placed in a segment of shared memory that is in the address space

of every process in the system, then there would be no need for a centralized TPM process : all of the server lookup could be done in a set of library routines that manipulate the shared memory segment.

A form of broadcast could also be implemented in a efficient way by writing the request to be broadcast into a shared memory buffer and giving the buffer ID of the request to all of the servers that participate in the event.

7 Future Plans

We are merging our existing event model with the nested transaction model such as in [1]. In the nested transaction model, a transaction can invoke a *subtransaction* which can invoke its own subtransactions, resulting in a tree of parent/child relationships. Each subtransaction tells the system if it desires to commit or abort, but for a subtransaction to truly commit, all of its ancestors must also decide to commit.

In our current model, a *subevent* can be called as part of processing an event. However, as far as the system is concerned, the child commits or aborts independently of the parent. Transactions can only be composed if they can be run in a parallel fashion, The advantage to be gained by merging with the nested transaction model is an increase in the ways transactions can be composed within code to form new transactions.

This new system will rely heavily on the availability of UNIX System V shared memory in order to reduce the cost of IPC and synchronization.

References

[1] J. Eliot B. Moss. *Nested Transactions : An Approach to Reliable Distributed Computing.* MIT Press, 1985.

DISTRIBUTED ON-LINE TRANSACTION PROCESSING SYSTEMS ON UNIX

Kirit Talati, Ph.D.
VISystems Inc.

Abstract:
A full function distributed on-line transaction processing system must distribute application functionalities over heterogeneous computer systems. VIS/TPS is the first such distributed on-line transaction processing (OLTP) system implemented on UNIX. This distributed OLTP system is based on logical interfaces such as are incorporated in the Systems Application Architecture (SAA) project. VIS/TPS supports both the client-server computing model as well as cooperative processing systems like APPC/LU6.2. Additionally, it addresses such common problems as data transformation and data access. This paper includes a comparative analysis of SAA-type solutions, the X/OPEN OLTP Reference Model and distributed OLTP systems like VIS/TPS.

Introduction

UNIX is a time-sharing operating system that is analogous to IBM's MVS/TSO and VM/CMS environments. Like MVS, UNIX is not conducive to transaction processing [1]. However, just as IBM's Customer Information Control System (CICS) executes under MVS [2], On-Line Transaction Processing (OLTP) systems can be built under a time-sharing operating system like UNIX. CICS, of course, performs the major functions of an operating system and, in that respect, it is an operating system in its own right. Even so, the attempt by some [3] to compare CICS to UNIX in terms of transaction processing potential is indefensible because UNIX is analogous to MVS but not to CICS.

The increasing industry acceptance of UNIX coupled with the rising demand for transaction processing has resulted in the development of a whole range of transaction processing, OLTP and distributed OLTP products that can execute under UNIX. Examples include transaction processing systems from database vendors like Oracle and Sybase, OLTP systems from hardware vendors like AT&T and distributed OLTP systems from software vendors like VISystems Inc. Distributed OLTP is understood here as a mode of application processing in real time in which messages from user devices are processed for functions such as data manipulation while distributing the processing of the application across multiple CPUs, whether the functions are on the same or on different computing systems.

Distributed On-Line Transaction Processing has become all but mandatory for corporate America as it moves from centralized to distributed processing. This shift is in good part a consequence of (a) the proliferation of powerful processors on platforms that span the whole spectrum from workstations to supercomputers and (b) the corresponding pressure to facilitate the total interconnection of corporate data processing resources. Major hardware vendors are attempting to address these requirements with various distributed solutions.

IBM is seeking to interconnect such incompatible systems as the PS/2, the RT/PC, the S/36, the S/370 and the AS/400 with its Systems Application Architecture (SAA) [4], a solution based on the backward integration of hardware and software technologies. Under SAA, applications can be structured in such a way as to be processed in a distributed environment so that the applications can share data and services even when they are executing on different IBM platforms.

In contrast to IBM, DEC has adopted a forward integration strategy based on a single (proprietary) operating system (VMS) that allows cross-system application development on its microcomputers, minicomputers, workstations and mainframes. DEC's implementation of a distributed OLTP under VMS provides an SAA-type strategy tailored to the DEC VMS environment.

UNIX-based systems can also interconnect corporate resources and may provide the ideal distributed OLTP. Since UNIX is used in a multi-vendor environment, a UNIX-based distributed OLTP must address heterogeneous computing environments.

X/OPEN, the international consortium of hardware and software vendors, recently released a working document on an OLTP reference model for UNIX-based systems that is analogous to SAA [5].

This OLTP model comprises three different semantic layers: resource (data, device, etc.), commit (transaction integrity mechanism in a distributed environment) and a transaction manager for the control and execution of distributed transactions. The model includes a protocol boundary that provides a generic application interface. This interface, in turn, is made up of a set of verbs that describe the primary function of the OLTP model. The protocol boundary is analogous to the Common Programming Interface, the Common Communications Interface and the Common User Access offered by SAA and the Application Programming Interface provided by CICS.

Distributed On-Line Transaction Processing Systems

As described earlier, distributed OLTP systems must distribute application functionalities across heterogeneous computer systems. Several technologies available today can be utilized to this end. The client-server computing model as well as widely used peer-to-peer communications protocols, like Socket and TCP/IP in a UNIX environment and LU6.2/APPC in the IBM environment, enable two or more programs to execute in independent environments while working cooperatively to perform a single task. With such facilities, it is now possible to build distributed OLTP systems - even under UNIX. Distributed OLTP systems in a UNIX environment must overcome four major obstacles if they are to be viable.

1. Virtual Terminal Support

Though UNIX supports character-oriented devices like asynchronous terminals, it provides virtually no support for block-oriented devices like the 3270 data stream and ATMs. Such facilities as Streams and Transport Level Interfaces can provide the required support for block-oriented devices but this support is not generally available on UNIX platforms. Third party vendors do provide interfaces to block devices but such interfaces are inadequate in an OLTP environment unless these are integrated with the OLTP system.

2. Distributed Data Management

There are virtually no distributed data management facilities under UNIX. Of late, however, UNIX database vendors are building distributed database management systems (DBMS) that are based on the relational model and on SQL standards. Integration of these DBMS with the OLTP environment is dependent on the emergence of applicable industry standards. The ISO OSI Remote Database Access method, which is based on a client-server computing model, has become fairly prominent but it is still two to three years away from becoming a standard. Applications executing in a distributed OLTP system under a multi-vendor UNIX environment have to be able to access data transparently without being restricted by the location, the nature and the format of the data. The DBMS and the Network Manager can locate and extract data. However, the need to update distributed databases while maintaining data integrity and recoverability is currently a major stumbling block. The ideal solution would be a two phase commit protocol but this has yet to be implemented in DBMS in distributed environments. The problem of data transformation is currently handled by building gateways and by using a distributed data dictionary - though a distributed data dictionary has yet to be implemented by most hardware and database vendors. These problems are not peculiar to UNIX but are inherent problems of distributed data management on heterogeneous computer systems.

3. Connectivity

As noted earlier, distributed OLTP applications on UNIX must be able to access data and services from non-UNIX OLTP systems such as CICS. Connectivity between UNIX and non-UNIX computer systems is now a viable option utilizing TCP/IP and LU6.2/APPC. Most UNIX vendors are planning to make these protocols available under UNIX.

4. Performance

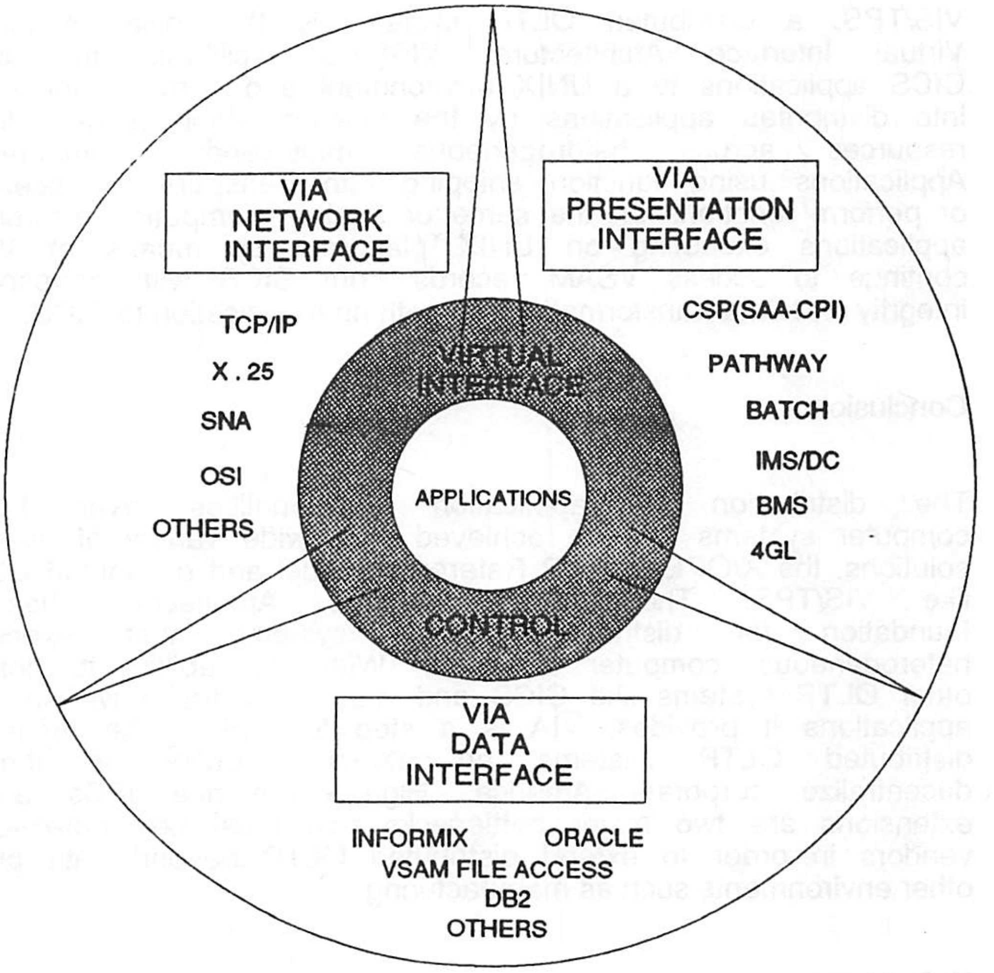
Performance in a distributed OLTP environment must be acceptable not only in data processing but in a manufacturing environment in order to use distributed OLTP to decentralize corporate America. A major component of UNIX, the scheduler, must be modified to support a real-time environment: the scheduler must be able to provide a deterministic or predictable response to an external event. A shared library and a high-performance Inter-Process Communications facility would also be helpful in UNIX in order to match the kind of performance given by CICS under MVS.

Even if a distributed OLTP under UNIX were available today, the most serious obstacles to its acceptance in the data processing market would be connectivity to existing non-UNIX platforms and the tremendous corporate and user pressure to preserve the vast investment in the "installed base" of applications and user skills. VIS/TPS is a distributed OLTP that addresses these issues of connectivity with non-UNIX platforms and offers migration tools for the installed base of applications. The proprietary Virtual Interface Architecture (VIA) on which VIS/TPS is based is described below.

Virtual Interface Architecture

IBM's Systems Application Architecture (SAA) and VISystems Inc.'s Virtual Interface Architecture (VIA) are two technologies that provide a distributed OLTP environment using logical interfaces. VIA extends the use of logical interfaces to allow the migration of current applications to a distributed environment by providing a bridge between the application's original environment and the new environments to which it will have access through standard connectivity technologies.

The figure on the next page shows the key components of Virtual Interface Architecture: the Virtual Presentation Interface, the Virtual Network Interface, the Virtual Data Interface and the Virtual Interface Control.



VIA Meta-Operating System

<u>Virtual Presentation Interface</u>

Masks the application's dependencies with respect to presentation services used in systems like CICS, IMS/DC, etc.

<u>Virtual Network Interface</u>

Provides a liaison between applications and the network connectivity enabled by industry standards. It provides functions such as data transformation, virtual terminal protocol and remote user access in a heterogeneous environment.

<u>Virtual Data Interface</u>

Provides a liaison between applications and the Data Base Management System (DBMS) and coordinates transaction management with the DBMS to provide data integrity.

<u>Virtual Interface Control</u>

Acts as a distributed resource manager by navigating networks, data and presentation, and by providing a logical interface between the application's old environment and new environments, thus forming a meta-operating system.

VIS/TPS, a distributed OLTP system, is the initial implementation of Virtual Interface Architecture. VIS/TPS facilitates the migration of CICS applications to a UNIX environment and converts these applications into distributed applications by the function shipping of data and other resources across heterogeneous, multi-vendor computer systems. Applications using function shipping can transparently access resources or perform functions on the same or another computing environment. CICS applications executing on UNIX platforms by means of VIS/TPS can continue to access VSAM records from CICS with corresponding data integrity and data transformation and with no modification to CICS.

Conclusion

The distribution of application functionalities over heterogeneous computer systems can be achieved in a wide variety of ways: SAA-type solutions, the X/OPEN OLTP Reference Model and distributed OLTP systems like VIS/TPS. The Virtual Interface Architecture has laid the foundation for distributed OLTP systems that execute across heterogeneous computer systems. With its ability to interface with other OLTP systems like CICS and with the virtual environment for CICS applications it provides, VIA is a step forward in the attempt to make distributed OLTP systems an attractive option in the drive to decentralize corporate America. High-performance IPCs and real-time extensions are two major bottlenecks that must be addressed by UNIX vendors in order to extend distributed OLTP beyond data processing to other environments such as manufacturing.

References

[1] "Camelot Perspective", Jeffrey Eppinger and Alfred Spector, Unix Review, January, 1989, pp. 58-67.

[2] CICS Application Programmer's Reference Manual, IBM, Version 1.7, Program # SC33-0241-0.

[3] "Cause for Concern", Shirley Henry, Unix Review, September, 1987, pp. 47-55.

[4] "Systems Application Architecture: An Overview", IBM Document Number GC26-4341-0, April 27, 1987.

Building a Transaction Processing System on UNIX® Systems

Juan M. Andrade, Mark T. Carges and Kurt R. Kovach
AT&T Bell Laboratories
Summit, NJ 07901

ABSTRACT

An on-line transaction processing (OLTP) system must provide high performance, as measured by transaction throughput and by response time, to a high volume of concurrent user requests from many terminals. It must provide this performance without sacrificing reliability. Building an OLTP system on a time-sharing operating system is particularly difficult since the needs of time-sharing users are very different from the needs of OLTP users. This paper discusses how a particular high performance transaction processing system has been built on a UNIX System V base. It discusses the transaction processing model chosen as well as a high level description of the implementation of the AT&T TUXEDO Transaction Processing System. The TUXEDO system has found widespread use within AT&T in applications that vary from forecasting and financial planning to provisioning and administration of long distance networks.

1. INTRODUCTION

Time-sharing operating systems, like the UNIX system, try to maximize the concurrent use of different system resources by a large number of users. Users are provided with independent working environments that allow them to perform very different kinds of work (e.g., compilation or text editing). Since system load is difficult to predict, the selection of job priorities is based on providing all users with a fair share of the machine resources with the goal of providing good response time to interactive commands.

The basic services provided by time-sharing environments may not fully satisfy the requirements of an OLTP System. An OLTP system generally performs a large volume of relatively simple, repetitive tasks that arrive in predictable patterns [Anderson] [Dwyer]. Users work in close cooperation to share information generally stored on one or more databases. The OLTP system must provide both performance and reliability which includes the guarantee of correct database updates and recoverability from events such as program, operating system and hardware failures.

The ability of the UNIX operating system to support OLTP and database management systems (DBMS) adequately has been questioned in the past [Stonebraker] [Henry]. Ideally, an OLTP system constructed on top of a general purpose operating system requires the operating system to provide a relatively small set of highly efficient services. To build an OLTP system in such an environment, one must:

- Understand the strengths of the operating system and use these services where appropriate.
- See where the basic mechanisms are provided but functionality is lacking and add the needed functionality.
- Augment the system with special purpose software where operating system services are unavailable or unsuitable.

This paper discusses how an OLTP system has been built on top of UNIX System V. The AT&T TUXEDO Transaction Processing System provides the functionality, performance, reliability and integrity required by OLTP systems built on medium-scale computers running UNIX System V.

Section 2 describes a model for building OLTP systems in UNIX environments. This model is based on a client/server approach. The architecture of the TUXEDO System is discussed in section 3. Section 4 outlines some directions for the TUXEDO System in distributed environments. Finally, section 5 gives a short summary of the ideas presented in the paper.

2. CLIENT/SERVER MODEL

When building an OLTP system, it is important to choose a model which supports the following characteristics:

- OLTP systems must be designed with performance in mind. Communication within the system must be efficient and require little overhead. To support high transaction rates, the system must also make efficient use of its resources (e.g., minimize idle processes).
- As the workload increases on a system, or as more users are added, an OLTP system's throughput should increase in proportion. Compared to time-sharing systems, transaction systems have much more predictable workloads and should use operating system resources more efficiently. Therefore, the workload of an OLTP system should be able to increase steadily while not heavily burdening the operating system for services.
- A transaction system should allow its users to model their applications according to their business needs. Certain transactions may be accessible, for example, only during certain hours of the day. An OLTP system should serve an application's needs and allow the system to change dynamically without disruption to the running application.
- OLTP systems must be reliable. There are many degrees of reliability. The ultimate in reliability is fault tolerance: a system continues to function with little performance degradation in the face of hardware and software failures. Many systems offer something not quite as grand as full fault tolerance. For example, in a UNIX environment, a system could monitor the processes accessing a database and respawn those that die abnormally.

An architecture well suited to UNIX transaction processing which addresses the above needs is one based on a client/server approach. In the TUXEDO System, clients are processes which gather and send input in the form of service requests to server processes which access DBMS resources on the clients' behalf. Typically, clients gather their input from end-users at terminals or workstations. Servers, on the other hand, are tailored for accessing DBMS resources in a fast, highly repetitive manner. In the TUXEDO System, servers have the following properties:

- Servers are daemon processes which continually accept requests and dispatch them to application routines, called services, for processing. It is the service routines which access DBMS resources.
- Servers are single-threaded and not re-entrant (i.e., a server is unavailable for processing until it finishes its current request and there is no system support for maintaining "state information" although an application may provide this support).
- Servers may advertise many services and one service may be advertised by many servers.
- Servers may act like clients and make service requests to other servers.

While it may take an end-user one or two minutes to enter the data for a service request on a screen, it usually takes a database server tens of milliseconds to access a DBMS and complete a request. Thus, in a typical client/server configuration, a small number of server processes can handle a very large number of end-users.

Clients and servers communicate in a fashion similar to remote procedure calls (RPC) [Birrell]. Once a client has gathered all the data for its request, it packages the data together with information about which service is desired and sends the request to the appropriate server. A transaction processing monitor (TPM) handles the routing of a client's request to an appropriate server process. When a client sends its request, it can either wait for the reply or asynchronously receive the reply at its convenience. When a server receives a request, it processes the request and sends a reply back to the client. A server may act as a client itself and send requests to other servers as part of fulfilling the original client's request. Figure 1 illustrates the client/server architecture with a process model showing where the different application and OLTP system entities exist. Note that application code exists in both client and server processes.

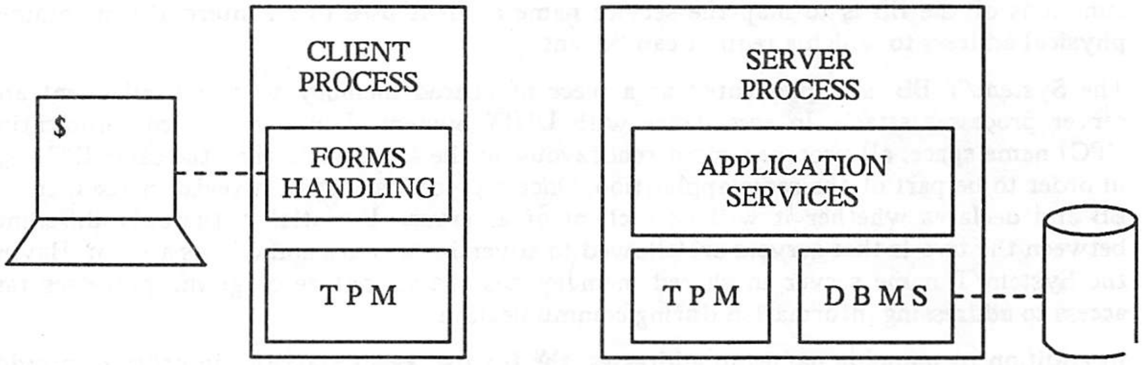


Figure 1. A Process Level View of the Client/Server Model

The client/server model emphasizes local autonomy, modularity, and data independence [Gray87]. The design methodology suggested by the model follows that of object-oriented programming languages: access to a particular database can be abstracted through a set of elementary or composite operations. These operations are implemented as a set of services that are offered by one or more servers. An elementary operation applies to a single context (i.e., a particular operation on a single DBMS), while a composite operation is a combination of elementary operations offered across different contexts (i.e., different operations across many DBMSs). Seen from this position, the client/server model provides the framework on which distributed transaction processing systems can be built (see section 4 for more details).

3. ARCHITECTURE

The AT&T TUXEDO Transaction Processing System has two main components:

- System/T, a TPM that controls and routes communication among a set of client and server processes and provides administrative capabilities for dynamically tuning an application.
- System/D, a transaction-oriented DBMS that provides an interactive SQL interpreter, an embedded SQL for the C language, and a library of functions for manipulating records one at a time.

This section concentrates on how each component has used UNIX System V facilities to build features for OLTP.

3.1 TUXEDO System/T: A Transaction Processing Monitor

System/T provides a means of communication among a cooperating set of processes in an OLTP application. As is described in more detail below, clients and servers communicate with the aid of the System/T name server in a "connection-less" manner. For OLTP applications, this style of communication makes efficient use of UNIX system resources, and scales well with respect to both increases in the system load and distributed environments. System/T also allows administrators to view their running application and tune it dynamically to suit their needs. The remainder of this discussion focuses on System/T's name server, communication paradigm and administrative capabilities.

Name Server

The heart of System/T is its name server, called the System/T Bulletin Board (BB). As a name server, the BB allows clients to communicate with servers by name. Clients send their requests to the name of the service they wish to invoke, rather than to a specific address. Clients do not know, nor do they care which server handles their request. One of the main functions of the BB is to map the service name a client uses to an internally maintained physical address to which a request can be sent.

The System/T BB is implemented as a piece of shared memory to which all client and server processes attach. In accordance with UNIX System V interprocess communication (IPC) name space, all processes must rendezvous on the same name (i.e., the same IPC key) in order to be part of the same application. Once a process attaches, it registers itself in the BB and declares whether it will be a client or a server. Essentially, the only difference between the two is that servers are allowed to advertise services and clients are not. Having the System/T name server in shared memory has the advantage of giving processes fast access to addressing information during communication.

In addition to mapping names to addresses, the BB also keeps statistics in order to provide load balancing when deciding where a client's request should be sent. Because one service can be offered by many servers, it is important that every request is sent to a server which can handle the request with minimal delay. Statistics are gathered every time a request is sent and also when a request is processed so that the state of the system is accurately represented. More precisely, if upon sending a client's request, System/T knows how many outstanding requests exist and to which servers they are destined, it can send the request to a server which is most likely to process it fastest. Because information is being read from, as well as written to the BB, consistency is guaranteed by a combination of user and system level locking.

Communication Paradigm

Clients and servers communicate using UNIX System V messages. UNIX System V messages have some very desirable properties for OLTP. First, messages have clear boundaries and can be treated as record-oriented objects (i.e., a server has no trouble differentiating among many clients' requests). Second, messages can be dequeued in other than strict FIFO order. This property can be exploited for implementing application defined priorities on requests. Third, messages are efficient to send among a set of unrelated processes (in the UNIX system sense) due to their connection-less nature. This last property turns out to be very important for OLTP systems and is worth a closer look.

The characteristic which sets the IPC subsystem apart from the rest of the UNIX system is its name space: names in the IPC subsystem are not part of the UNIX file system name space. Unlike file descriptors, the handle used to identify a message queue is independent of any process. Any process that can obtain a message queue's handle can send a request directly to that queue. Clients do not have to perform an "open" for every server with which they may ever want to communicate. Much efficiency is gained by using a single system call to send a request, as opposed to a paradigm where an open and close are performed around the communication. This is especially true for OLTP applications which are exemplified by large numbers of short interactions between client and server: the costs of opens and closes, or of retaining open file descriptors for every server in the system becomes prohibitive. In the System/T BB, service names map to message queue handles which the System/T routines within clients use directly in the message send system call (i.e., **msgsnd(2)**). It is important to note that using this type of communication, processes have no connection during communication. UNIX System V, in essence, provides a reliable, flow-controlled datagram facility perfectly suited to fast client/server interactions.

Message queues are used by servers in one of two ways. Either each server dequeues its requests from its own private message queue or a set of servers that all offer the same services share a single queue. The two methods are analogous to the style of queueing performed in most supermarkets and banks, respectively. With the single queue per server model, applications can enforce single threading all requests of a certain type through a particular server. For example, an OLTP application's performance would suffer greatly if many clients simultaneously made requests for an ad hoc report which required a scan of the entire database. By accessing such a service through a single server, the set of such requests is effectively single threaded. With the shared queue approach, load balancing is automatically performed by the servers since they will dequeue requests as soon as they are free to do so. Thus, for high volume transactions, an optimal balance of the work load is maintained while minimizing idle time in server processes. By allocating services to servers and servers to queues appropriately, applications have much freedom in designing a system which suits their needs.

Figure 2 shows clients and servers in a System/T configuration along with a BB and each process' message queue. Note that each process is attached to the BB shared memory.

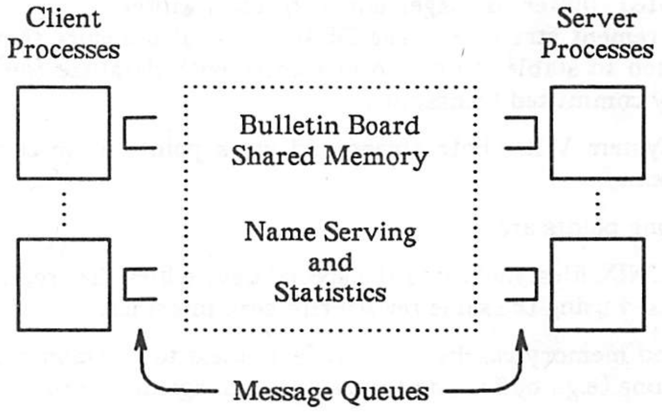


Figure 2. Clients and Servers in a System/T Configuration

Administration and Recovery

Because all name serving information and statistics are kept in the BB, system administration capabilities for monitoring and tuning a running application are easily provided. An administrative process, available as part of System/T, gives OLTP administrators a "window" into the information kept in the BB. Such information as the current and average request queue lengths, the number of requests performed by each server and service, and the current service being performed at a server can be shown on demand. Using this information, an administrator may decide to add or delete servers. Also, if an application wishes to allow or disallow the use of certain services while the system is running, facilities are available for suspending and resuming service advertisements.

Towards reliability, System/T provides a mechanism for restarting failed server processes. While a client process is awaiting its reply, a server process could die for some reason. System/T detects the failure and spawns a new instance of the failed server. When the new server starts, it is able to handle the client's request that was in progress when the old server died. In addition, since message queues are independent of processes, all queued requests are available to the new server. Information is kept in the BB about which servers are working on behalf of which clients. It is transparent to client processes that a failure occurred.

3.2 TUXEDO System/D: A Transaction-Oriented Database Management System

At the heart of the DBMS portion of an OLTP system, the file system must provide reliability, high performance and other features necessary to support data access and storage. Some of these other features are:

- Very large files must be handled efficiently. Creating files which span multiple disk partitions must be supported. The number of physical disk accesses required to retrieve (modify) information must be minimal and the overhead of accessing data in any block must be independent of its position in the file.
- Disk space must be managed effectively: storage for database files should be allocated in physically contiguous extents to reduce seek and latency for sequential access. The logical block size should not be fixed for all files but should be optimized for individual files.
- The database manager must control the buffer cache. More specifically, the flushing of data to stable storage and buffer replacement algorithms should be controlled by the DBMS. Normally the operating system does not have enough information to do DBMS-oriented buffer management correctly: different file types may warrant different replacement strategies. The DBMS must also ensure that appropriate blocks have been written to stable storage to guarantee both database consistency and recoverability for every committed transaction.

UNIX System V has both strong and weak points when considered as a base for an OLTP file system.

The strong points are:

- All UNIX files, including the special device files that reference "raw" disk partitions, are accessed using the same read/write/seek interface.
- Shared memory can be used for fast access to common data among processes on the same machine (e.g., buffer and transaction management information).
- Fast synchronous I/O is provided through the raw mode interface to disk partitions.

However, the use of regular UNIX files to support an OLTP file system has several potential problems [Stonebraker]:

- The operating system controls the flushing of data in the buffer cache, hence database consistency after a crash is never guaranteed. Although use of the UNIX synchronous write option solves the consistency problem, the performance penalties mentioned below limit its usefulness in high performance systems.
- An LRU replacement strategy is always used. This strategy may not lead to optimal performance for database systems that require sequential access to blocks that will be cyclically reused.
- The block size is fixed on a per file system basis, not on a per file basis.
- Consecutive blocks of the same file are not necessarily written to contiguous areas on the disk. Therefore, sequential file access may cause excessive disk head movement.
- Indirection, sometimes even double or triple indirection, is used for access to all but a few of disk blocks in a UNIX file system. The performance penalty caused by the extra I/Os required can be substantial (e.g., writing a direct block requires 2 I/Os and an indirect block 3 I/Os).

System/D provides a reliable database environment for OLTP applications by implementing a separate, user level file system which is independent of the UNIX file system. The file system is designed for storing database and transaction log information on a privately managed set of raw disk partitions. Since all I/O is synchronous and bypasses the UNIX buffer pool, data from committed transactions is written to stable storage providing database consistency and recoverability. Using raw disk partitions also avoids the performance penalties caused by indirect blocks. A buffer pool cache in shared memory minimizes the number of I/O requests and provides buffer replacement strategies optimized for the DBMS. Space allocation is extent-based with block sizes varying from file to file. Given the UNIX system's preemptive scheduling, consistency of shared memory data is guaranteed by a combination of user and system level locking.

System/D uses a redo/no-undo strategy [Bernstein], similar to IMS Fast Path [IMS], with updates logged to separate devices before being applied to the database. In the case of system failure, recovery is very fast since the system only has to redo the updates of committed transactions that are not in the database. Recovery from a disk crash involves restoring the disk to a previously "saved" version and applying logs accumulated since the saved version was created. Back-ups can be taken with the system either off-line or on-line, increasing system availability. System/D always performs an automatic warm restart when the system starts.

Consistency is provided through the notion of a transaction. A transaction may be started and ended in an application service routine as discussed above. The application service can generate several updates and then commit the transaction. On the other hand, a transaction may be started in one service routine and serially delegated to other service routines, one of which has the responsibility for ultimately committing the transaction. Updates are not visible to other transactions until after the commit point. The system provides concurrency and deadlock detection mechanisms. Concurrency is normally controlled through a two-phase locking strategy but several other consistency modes are also provided [Gray78].

The database can be accessed through interactive SQL, SQL embedded in C, or a function-oriented manipulation language that provides record at a time access. A variety of access methods including heap and FIFO files, hash files and B-trees are available.

In summary, System/D provides high performance access in a reliable database environment through:

- The extensive use of caching (using shared memory) optimized for DBMS access;
- Extent-based disk allocation;
- Support for multiple consistency modes for database transactions;
- Allowing programmers to navigate through a database with a record-oriented manipulation language when necessary.

4. DISTRIBUTED ENVIRONMENTS

The TUXEDO System architecture provides an open OLTP environment on UNIX systems in which applications are free to call operating system functions (e.g., file management) and other software packages (e.g., different DBMSs). The extension of this architecture to a distributed environment will provide applications with a single communication tool across different environments, and the possibilities for dealing with servers working on databases resident on different machines and with servers accessing heterogeneous DBMSs.

Connectivity through a LAN (or a WAN) to other computers running UNIX System V is an important goal for the TUXEDO System. To operate in a distributed environment, the name-serving part of the System/T BB (see section 3.1) will be replicated and, in addition, the algorithms for balancing the system load will be modified to take distribution into account. Some of these new mechanisms have been implemented already in a version of the

TUXEDO System that runs on a loosely-coupled multi-processor computer.

Another type of distributed connectivity is to mainframe machines running other operating systems and transaction systems. The best example is probably the connection to CICS through LU6.2. The client/server approach appears to be ideal for building distributed applications working in heterogeneous environments, but currently transaction processing standards for heterogeneous interoperability are not stable. ISO is defining a TP protocol [ISO-TP] which closely resembles IBM's LU6.2, and X/OPEN has recently begun work on defining standard interfaces for distributed transaction processing.

Section 2 provided a description of the client/server model implemented in the TUXEDO System, and figure 3 presents an extension of the model to the distributed case. This figure shows the different functional relationships between the main components of a distributed transaction processing system (DTP). The three main components are: the application which provides the service abstraction, the DBMS that controls local data and provides local concurrency control and recovery management, and the TPM which provides mechanisms for remote service activation, remote server load balancing, distributed execution control, and global transaction coordination. These facilities within a client/server model provide application programmers with a distributed model that emphasizes local autonomy, modularity, performance, data independence, and location transparent access to remote databases.

Each DTP component interacts with the other components through well defined interfaces. For example, the application uses SQL to interact with the DBMS and a special set of TPM primitives to interact with another application. A TPM interacts with another TPM through a standard communication protocol. Finally, the interaction between the TPM and the DBMS implements a two-phase commit protocol ([Gray78] and [Bernstein]) as explained below.

The main goal of a DTP system is to allow a transaction to span multiple servers or sites while retaining its atomic properties. A distributed transaction, in this DTP context, corresponds to the execution and coordination of one or more services, each accessing possibly different DBMSs, as a logical unit of work. This logical unit of work should behave as a single global transaction (i.e., either all actions of the transaction have their expected effect or no action is carried out).

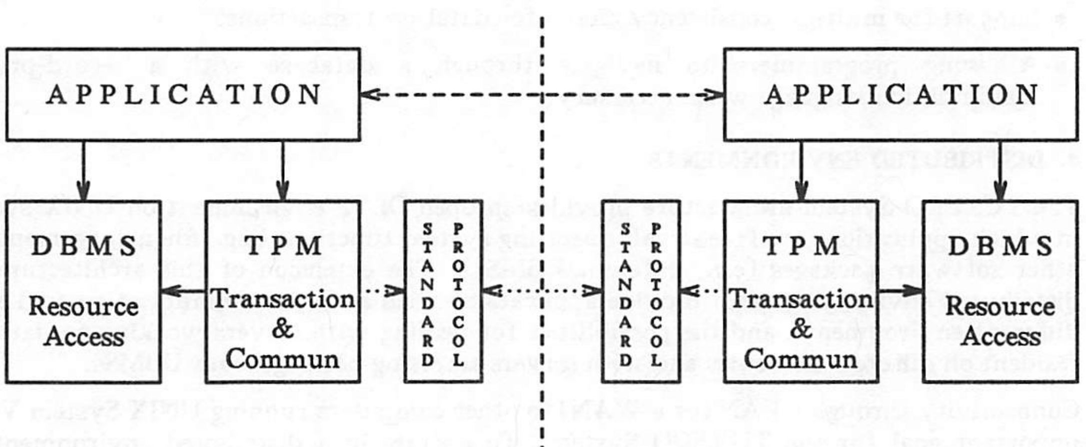


Figure 3. Functional Model for Distributed Transaction Processing

A DTP distributed transaction is composed of two parts: a global part controlled by the TPM and a local part controlled by the local DBMS. A single global transaction is mapped to possibly many different local transactions depending on how many different DBMSs are involved in the distributed transaction. Global transaction identifiers and the global decisions concerning the commitment of a transaction are managed by the TPM. Transaction coordination and recovery is controlled with a two-phase commit protocol, in which there is the notion of a coordinator and a set of participants. The coordinator instructs the participants to commit or abort a transaction. The same protocol is followed between the TPM and the local DBMS.

Thus, this DTP transaction model imposes three requirements on a DBMS. First, a DBMS must be able to perform a two-phase commit protocol.[1] Second, a DBMS must support the notion of a "foreign" transaction (e.g., to accept global transaction identifiers generated by TPM). Finally, a DBMS must not make decisions (abort or commit) on foreign transactions that were found in a pre-commit state at recovery time. This decision can only be made by the transaction's coordinator TPM, and for this reason, the DBMS must be able to distinguish foreign transactions from local DBMS transactions.

In the distributed environment described here, service routing tends to be data dependent. For example, in a banking application the account file could be horizontally partitioned by ranges of account numbers, and then each partition could be stored on a different computer. The services offered on each partition could be the same (e.g., credit, debit) but because these services could be requested from any computer in the network, the application code would have to control the location dependencies. There are different ways of providing an application location independence in this case; for example, associate a service with a field (e.g., account number) on which a partitioning criteria applies. The TPM can perform data dependent routing transparent to applications by looking at the field's value and then applying the corresponding partitioning criteria to find the right service location.

Finally, the implementation of distributed SQL queries is considered important for closely clustered distributed environments. The DTP facilities discussed here provide a good base for homogeneous and heterogeneous distributed SQL DBMS environments. However, to provide such interaction with heterogeneous DBMSs, a standard protocol for SQL remote access is required. While work is being done by ISO RDA [ISO-RDA] to define a protocol for remote database access, this protocol is not yet in final form.

5. SUMMARY

An architecture that satisfies the requirements of OLTP has been presented in this paper. The AT&T TUXEDO Transaction Processing System is proof that OLTP is possible in a UNIX environment without requiring *any* changes to the operating system. The architecture of the TUXEDO System adapts well to distributed environments and to tightly-coupled or loosely-coupled multi-processors. The TUXEDO system has been used in a variety of AT&T applications, including support of long distance networks both domestically and abroad, and has been available in the commercial market since late 1988.

1. Most likely the "presumed abort" version of two-phase commit [Mohan] will be required. This version of the protocol is efficient with respect to the number of messages sent between participants. Also, ISO TP is leaning towards this version of the protocol.

6. ACKNOWLEDGEMENTS

We would like to thank the members of the TUXEDO project. Also, D. Shasha, T. J. Dwyer, and H. Schottland provided valuable comments. Finally, C. Samanta provided us with the initial stimulus to write this paper.

7. REFERENCES

[Anderson] K. J. Anderson, "Bucket Brigade Computing," UNIX Review, August 1985, pp. 58-64.

[Bernstein] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, 1987.

[Birrell] A. D. Birrell, and B. J. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, 2/1, February 1984, pp. 39-59.

[Dwyer] T. J. Dwyer, "TUXEDO Transaction Processing System For 3B Computers," AT&T Technology, 3/1, 1988, pp. 40-45.

[Gray78] J. Gray, "Notes on Database Operating Systems," Research Report RJ2188, IBM, February, 1978.

[Gray87] J. Gray, "Transparency in its place," UNIX Review, May 1987, pp. 42-50.

[Henry] J. Shirley Henry, "Cause for Concern," UNIX Review, September 1987, pp. 47-55.

[IMS] IBM Corp., IMS/VS Version 1 General Information Manual, form no. GH20-1260-12, 1984.

[ISO-RDA] ISO/JTC 1/SC 21, "Information Processing Systems -- Remote Database Access," July 1988.

[ISO-TP] ISO/TC 97/SC 21 N 2274, "Information Processing Systems -- Open Systems Interconnection -- Distributed Transaction Processing -- Part 3: Protocol Specification," March 1988.

[Mohan] C. Mohan, B. G. Lindsay, R. Obermarck, "Transaction Management in the R* Distributed Database Management System," ACM transaction on Database Systems, December 1986, vol. 11, no. 4, pp. 378-397.

[Stonebraker] M. Stonebraker, "Operating System Support for Database Management," Communications of the ACM, 24/7, July 1981.

High Availability in a UNIX Transaction Processing Environment.

W. Stephen Adolph
MPR (Microtel Pacific Research Limited)
8999 Nelson Way
Burnaby, B.C. Canada
V5A 4B5
604-293-5360

ABSTRACT

The reliable management of the Digital Sevices Transmission Network (DSTN) is extremely important to the British Columbia Telephone Company (BC Tel) which has identified a need to increase the availability of the UNIX based Digital Support System (DSS), the DSTN network management system.

The perceived lack of high availability and reliability are two impediments to the acceptance of UNIX as a platform for supporting transaction processing. At MPR (Microtel Pacific Research Limited) we have been working to develop a high availability platform for DSS. While our specific solution for BC Tel is based on existing hardware architectures, a set of features for enhancing the UNIX kernel to support high availability will be proposed.

KEYWORDS

Fault Tolerance, Transaction Processing, Network Management, UNIX.

1. INTRODUCTION

The installation of digital transmission facilities and deregulation allow telephone operating companies (telcos) such as BC Tel, to offer new and competitive services to their customers. In the fall of 1989, BC Tel will inaugurate the Digital Services Transmission Network. BC Tel and its customers will be able to dynamically allocate and manage bandwidth throughout BC Tel's digital transmission network to meet their changing needs.

The Digital Support System (DSS) is the primary tool BC Tel uses to manage the digital transmission network. The growing importance of the digital transmission network to BC Tel's revenue and the offering of DSS services to external customers requires DSS to provide a higher quality of service. The current DSS implementation only provides support for one processor and for only one copy of the system data base.

The objective of the high availability feature for DSS is quite simply to provide a higher level of system availability and data base performance than what is provided by the current release of DSS. Telephone operating companies are extremely conservative and therefore reluctant to install new or unconventional hardware platforms into their operating environment. Our project charter restricted us from taking advantage of existing fault tolerant processors and therefore we required to build a fault tolerant platform by inter connecting conventional processors.

The focus of this paper is specific to the requirements of DSS, however, we will extrapolate our work to include recommendations for enhancing the UNIX kernel to provide a simple and portable mechanism for supporting highly available systems.

2. DIGITAL TRANSMISSION MANAGEMENT AND TRANSACTION PROCESSING.

DSS supports a large interactive data base that describes the network it is managing. DSS data base objects are nodes, links, paths, maps, and schedules. Nodes are physical transmission network elements such as digital cross connects and multiplexers. Links are the 24 channel 1.5megabit/second trunks called DS1s that interconnect the nodes. Paths are the logical assignment of an end to end assignment of bandwidth through the physical network. A path can be constructed from one or more of the 64kbit/sec channels that make up a DS1. A map is a collection of paths that can be scheduled in and out of service.

There is a reluctance in the telephone industry to admit network management tools, such as DSS, are transaction processing systems. The tight response time requirements of these systems usually lead their designers to classify them as real time systems. In fact, there is frequently no activity carried out by these systems that qualifies as a real time activity.

A typical DSS user transaction involves the creation of a path through the network. The DSS searches its data base for the appropriate DS1s to find and reserve sufficient bandwidth. If the path requires multiple hops, then bandwidth must be available on each of the hops, otherwise the transaction will fail and all previously allocated bandwidth released (atomic unit). This is analogous to reserving seats on an airline flight.

One important attribute of a transaction processing system that DSS does not have is a sophisticated system for assuring reliability and availability. This stems from DSS being a UNIX based system for which availability and reliability were not primary design goals. Our work is to provide a highly available UNIX transaction processing environment.

3. FAULTS AND AVAILABILITY.

A system may become unavailable because of scheduled events or unscheduled events. Unscheduled events that could make a system unavailable to a user community originate with faults. A fault is a deviation from normal expected system behaviour. A system may become unavailable because of hardware faults, software faults, or operator faults.

Increasing system availability requires the system to be more tolerant of faults that would otherwise cause a system failure.

A fault tolerant system is one whose architecture prevents a system failure after a fault has occurred within the system. The architectural attribute that makes a system fault tolerant is protective redundancy. Hardware and software elements that are critical to the correct function of the system are replicated such that a fault cannot cause a system failure. When a fault occurs, the system reconfigures itself to isolate the faulty component from the rest of the system.

A fault tolerant architecture can be used to make a system highly reliable or highly available. A highly reliable system is one that will produce a correct result in the event of a fault. Applications that require high reliability include industrial process control systems and aircraft fly-by-wire systems. These systems have redundant elements execute the same copy of a program in parallel and then compare the results. When there is a discrepency in the comparison the system is reconfigured by voting the faulty element out.

A highly available system is one that will remain up and running in the event of a fault. Applications that require high availability would include on line transaction systems such as automated tellers and airline reservation systems. These systems have redundant elements that are used to replace faulty elements. In some cases, the redundant elements can operate in parallel and share the processing load. When a fault occurs, the system will remain available but with degraded performance.

4. ARCHITECTURES FOR FAULT TOLERANT SYSTEMS.

We used the model proposed by Lampson and Sturgis [KAME83] as a framework for understanding the requirements of highly available systems. The model constructs a hierarchy of more fault tolerant abstract machines as an approach to implementing atomic transactions. Transactions are assumed to consist of a sequence of read and write commands sent by a client to the file system. The model divides events into two categories, desired and undesired. Undesired events are further subdivided into expected and unexpected.

The model categorizes system hardware into disk storage, communications, and processors. Disk storage is modelled as a set of addressable pages, where each page contains a *status* which may be good or bad, and a block of *data*. A processor can communicate with the disk by two operations *Put* and *Get*. The normal expected effect of these operation is page of data is correctly written or read from the disk. The expected undesired effect is soft I/O error and the unexpected undesired effect would be an undetected error.

A reliable system is constructed from a hierarchy of more reliable virtual machines. For example *stable storage* is designed to provide operations equivalent to the *Put* and *Get* while eliminating undesired expected events such as soft errors and spontanous decay that are associated with the physical disk.

Figure 1 shows how stable storage is constructed from an intermediate abstraction called *careful disk storage* which eliminates soft disk errors. Careful disk storage uses *CarefulGet* and *CarefulPut* to repeatedly perform *Get Put* operations until a good status is obtained or until the operation has been tried *n* times.

Stable storage eliminates hard disk faults by creating a *stable page* which consists of a pair of careful disk pages one of which is designated the primary. A *StablePut* is an atomic operation which performs *CarefulPuts* on a pair of *careful pages*.

This hierarchy of virtual machines provides a simple and elegant structure for constructing highly available systems.

4.1 Evaluating Fault Tolerant Architectures for High Availability.

Fault tolerant architectures can be evaluated in terms of coupling between system components and the system reconfiguration strategy. Coupling is a measure of how tightly system resources are shared and how dependent system elements are on other system elements. A tightly coupled system is characterized by direct sharing of resources. A loosely coupled system is characterized as a system where a client must access resources through a resource server and communications between the server and client is facilitated by a general purpose communications network. The following table identifies the tradeoffs between loosely and tightly coupled systems [ADOL89a]:

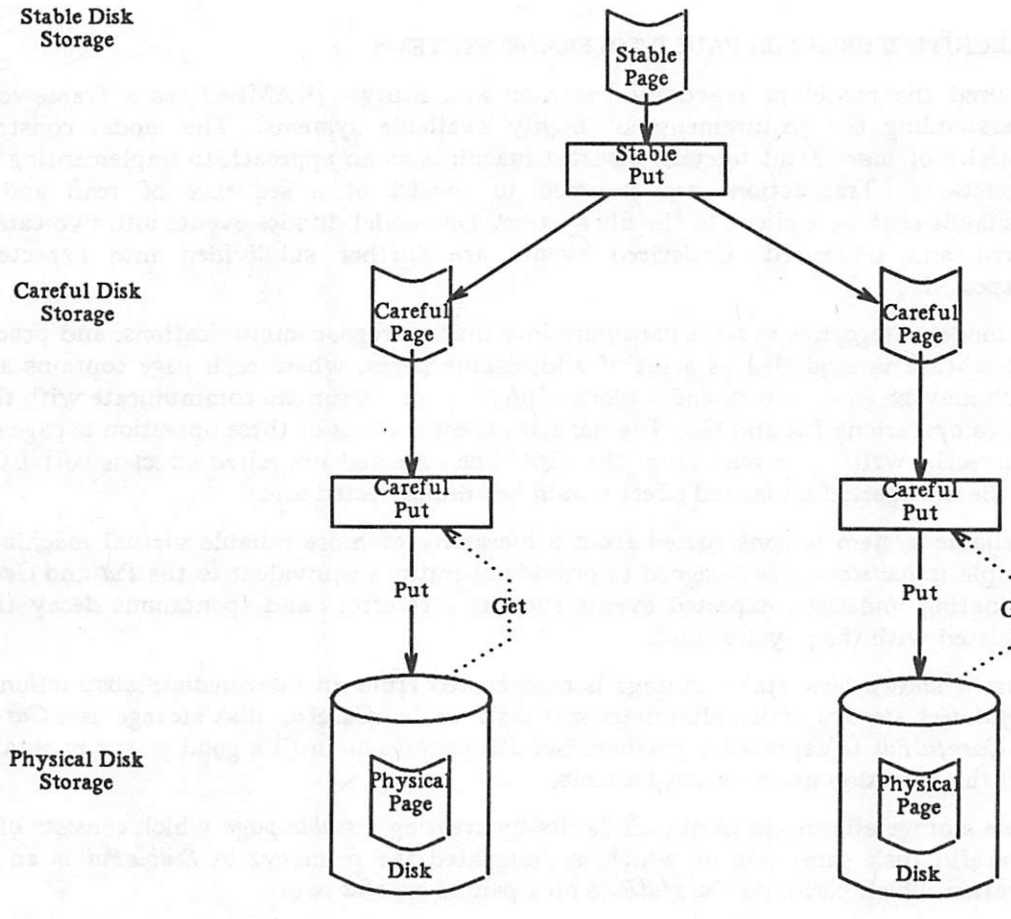


Figure 1. StablePut Operation Hierarchy.

	Loosely Coupled	Tightly Coupled
Communications Overhead	High because all resource access is done over the communications network.	Low because all resources are locally accessible.
Configuration Dependency	Low because elements are separated by the communications network. Hardware changes are masked by the communications network.	High because elements must be plug compatible and have relatively the same degree of performance. A change in system hardware has a significant impact on other hardware elements.

Resource Sharing	Low because resource request must be serviced over the communications network.	High because resources are directly accessible by client.
Inter Processor Synchronization Requirements	Low because resource server can arbitrate access to resource.	High because processors must negotiate access to the resource.

The reconfiguration strategy can be expressed as the granularity of the recovery method. A fine grained recovery strategy will attempt to minimize the elements placed out of service during a recovery. A coarse grained recovery strategy could end up placing entire systems out of service. The following table identifies the tradeoffs between coarse grain and fine grain reconfiguration strategies:

	Coarse Grain Strategy	Fine Grain Strategy
Complexity	Low, because of the limited diagnosis required to reconfigure the system. A simpler reconfiguration strategy makes the fault handling software more reliable.	High, because extensive diagnosis is required to isolate the faulty element, and then select the appropriate reconfiguration strategy.
Scope	A fault puts large units of systems resources out of service.	Only those elements directly affected by the fault are put out of service.
User Impact	High, most likely the user would be forced off the system and be required to log in again.	Low, many faults could be relatively transparent to the user.

5. ARCHITECTURES FOR HIGHLY AVAILABLE SYSTEMS.

5.1 Clusters

A cluster type system is a tightly coupled system configuration where system resources can be shared by several processors [ADOL89b]. If a shared resource, such as a disk fails, then the master system would mark the resource out of service and continue normal operations with the remaining resources. Resource sharing permits the use of a finer grained system management strategy. Failure of the master system's processor causes the slave system to take over processing. Figure 2 is an example of a cluster architecture.

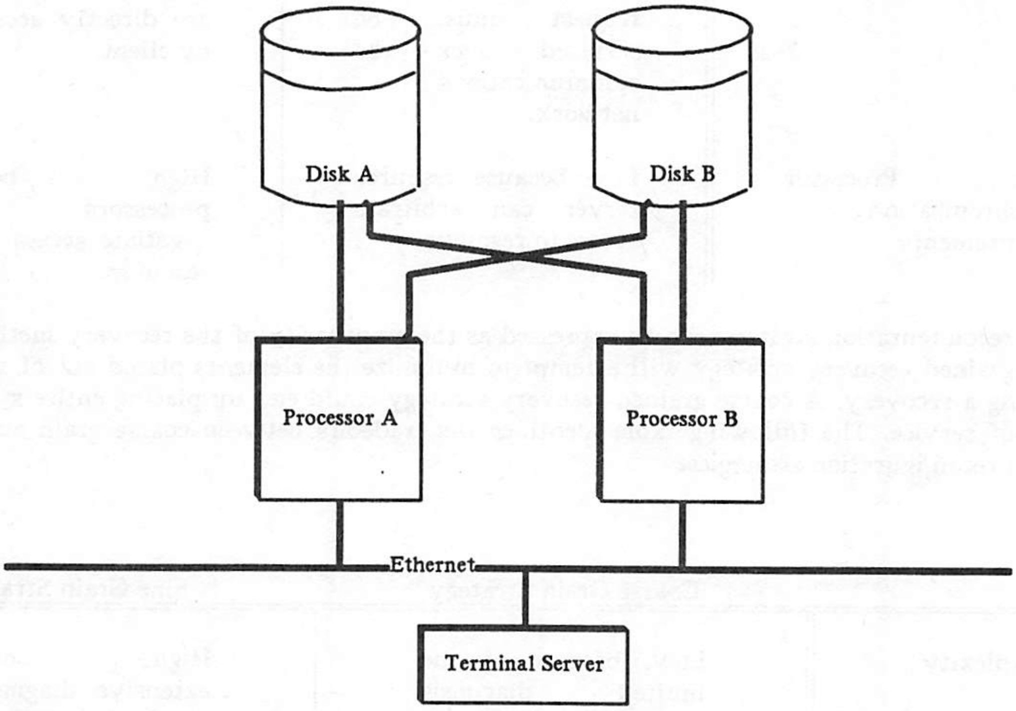


Figure 2. Example Cluster Architecture.

The main advantage of a cluster type solution is that it permits efficient utilization of system resources by maximizing resource sharing. All critical system resources are local to the processor accessing them. A processor would not be required to access the data base across a communications network. The degree of resource sharing is limited by UNIX because it does not support dynamic dual porting of disks.

The cluster solution as described can only practically support two processors and this raises a problem known as Byzantine disagreement. If a fault is detected, then it may be impossible to properly reconfigure the system because each processor has half the votes in the configuration. This problem can only be properly resolved by either including a third processor in the system to vote out the faulty processor, or by an external hardware arbitrator.

A cluster solution can be expensive in terms of the number of disks that need to be acquired. In addition to the shared disks acquired for the data base, a private disk for each processor would have to be acquired because the operating software on each processor will require a disk that it can write to (eg. paging and swap areas).

5.2 Server Based Systems

A server type system is a loosely coupled system where server processors provide access to system resources for application processors. The file server is architecturally similar to the cluster configuration with dual ported disks shared between two processors. Figure 3 is an example configuration of a server architecture.

The advantage of a server configuration is the de coupling of the resource server from the application processor. This permits the use of a coarse grain reconfiguration strategy and minimizes the disruption to user logins. The loose coupling between system elements means

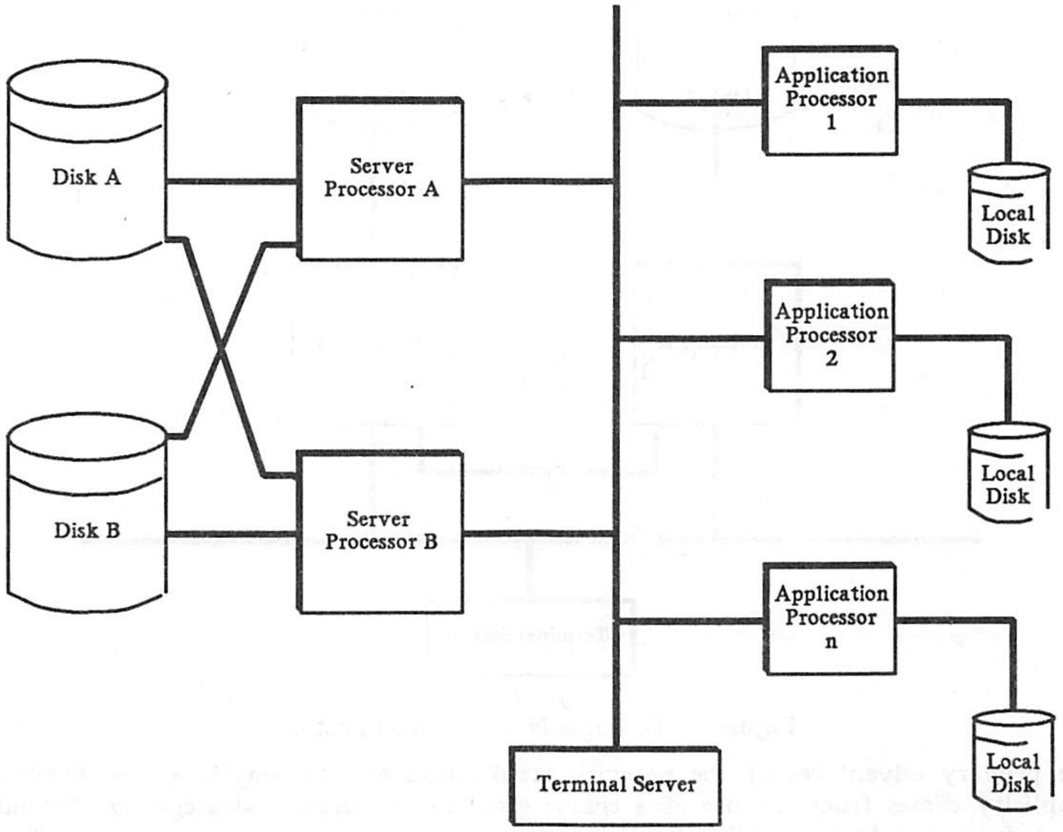


Figure 3. Example Server Architecture.

that individual processors could be modified or replaced without affecting other system elements.

A significant disadvantage to this architecture is that all data base accesses are across the communications network. The server based architecture is an expensive solution in terms of hardware resources required to implement it and has potentially low data base performance.

5.3 Network Based Architecture

A network type system features a moderate level of coupling where complete and independent systems are connected by a communications network. This configuration has attributes of both the tightly coupled cluster systems and the loosely coupled server systems. Each processor has a local set of resources and can have the other processor serve its resources over the communications network. The processors cooperate over the communications network to provide the high availability feature. Resources are shared by having each system act as a server for the other system. This low degree of resource sharing supports a coarser grained system management strategy. Figure 4 is an example of a network architecture.

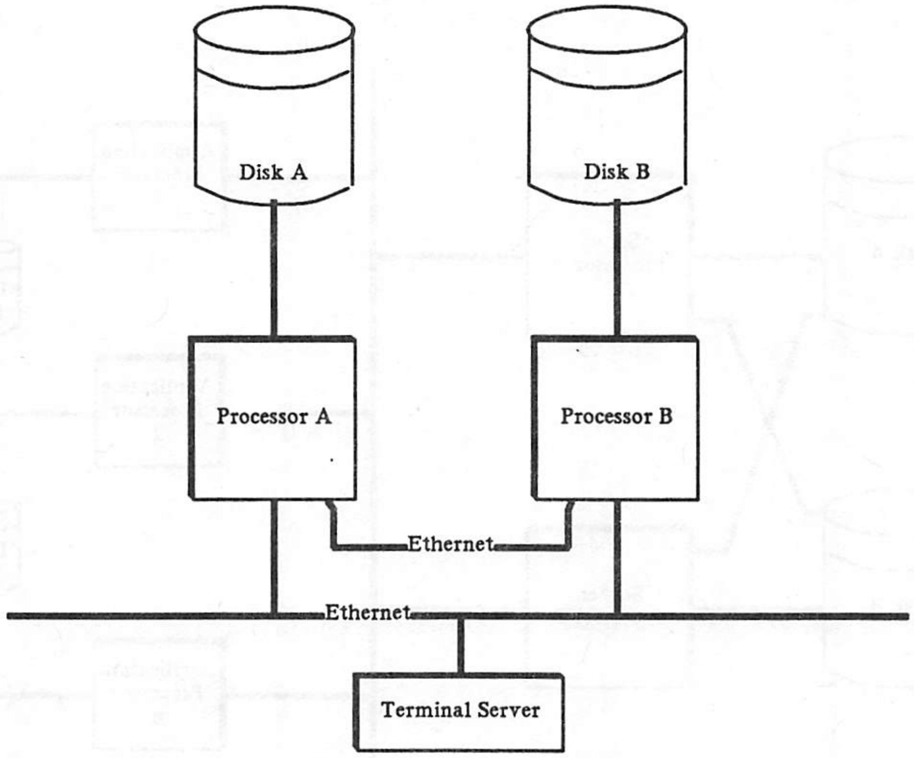


Figure 4. Example Network Architecture.

The primary advantages of the network architecture are its simplicity and flexibility. Simplicity comes from the use of a coarse grain reconfiguration strategy and flexibility comes from the loose coupling between the master and the slave processors. Different hardware could be used to implement each node in the network configuration.

The main disadvantage of this architecture is every disk write (transaction update) made on one processor, must be transmitted over the communications network to another. Resources cannot be utilized as efficiently as they could in a cluster solution because they are not shared by all processors. Downloads of outstanding transactions could place a significant load on the communications network when a processor is configured into the network (eg. booted).

The network configuration was chosen as the processor architecture to support the DSS high availability feature. The network configuration was not considered to be superior to the other processor configurations presented, but rather, it best satisfied the BC Tel requirements.

6. SUPPORT FOR RECONFIGURATION MANAGEMENT

Each processor participating in DSS maintains a processor configuration data base which is used to record the identity and the state of the other processors. When a processor is configured into the DSS system, it will begin broadcasting "I-AM-UP <node-id>" messages to all nodes recorded in the processor configuration data base.

When a node receives an "I-AM-UP <node-id>" message it will mark that node as up in its processor configuration data base. If a node does not receive an "I-AM-UP <node-id>" message within a pre defined period of time, then a processor will mark that node as down and begin a recovery.

To improve fault detection, a suicide message is added to this simple watchdog protocol. When a processor detects that it is failing, then it broadcasts an "I-AM-GOING-DOWN <node-id>" message. Reconfiguration is not delayed by requiring the other nodes to time-out on the "I-AM-UP <node-id>" watchdog.

This simple watchdog mechanism could be added to the UNIX kernel without loss of portability and the suicide message could be issued as part of the panic crash procedure. The kernel should not be distorted to support a specific recovery strategy because this would constrain the processor architectures that can be used to implement high availability.

7. A HIGHLY AVAILABLE DATA BASE.

The first release of the DSS high availability feature will only support a master/slave processor configuration and will not support load sharing. This greatly simplifies management of the online data base because it relieves us of having to implement the complex locking protocols required for synchronizing access to redundant copies of the data base. In keeping with the UNIX philosophy, we wanted to provide a simple mechanism for creating a highly available data base.

Our strategies for providing a highly available data base were influenced by System R [GRAY81]. Two copies of the data base are maintained, a primary copy and a shadow secondary copy. The secondary copy and the data base journal file are saved on the slave processor. Transaction updates are recorded into an update packet and written to the log file. The transaction is not considered committed until its update packet is written to the log. A writer process reads the log file and updates the secondary copy of the data base. The advantage of this scheme is that it minimizes pathological software faults from corrupting both copies of the data base because data base updates are executed by different code threads. In a master/slave processor configuration, one could model our primary data base as a volatile in memory data base with transactions flushed to the stable secondary data base.

Logging would be disabled when the system is running on only the master processor. When the slave processor is configured into the system, then a checkpoint is made of the master data base and transaction logging re enabled. The checkpoint data base is copied to the slave and the subsequent transactions are then applied to the checkpoint data base.

The UNIX block driver could be modified to provide support for disk shadowing. It would be desirable to define a logical disk volume so disk pages can be written relative to a logical block address to permit the mixing of different disk devices within the shadow set.

8. SUMMARY

UNIX's strength is its portability and any enhancement made to UNIX to support high availability must not prevent it from supporting a wide range of processor architectures. The addition of the following features to the UNIX kernel would provide support for high availability in a UNIX transaction processing environment and would not reduce its portability:

1. Processor Configuration Data Base,
2. Panic Crash Suicide Note,
3. Kernel Watchdog Timer, and
4. Disk Shadowing.

These represent the minimal set of primitive functions that if added to the UNIX kernel will provide a foundation for the development of highly available UNIX systems. The key element of any approach for providing high availability must be the preservation of the UNIX philosophy of design simplicity and portability.

9. ACKNOWLEDGEMENTS

I would like to thank George Myers, Ross Parker, and Dave Ashworth for their time and assistance.

10. REFERENCES

[ADOL89a] Adolph, W. S., "A Statement of Requirements for the DSS High Availability Feature," MPR Technical Report DSS-RA-890119, Microtel Pacific Research Limited, Burnaby, B.C.

[ADOL89b] Adolph, W.S., "An Evaluation of DSS Hardware Platform Alternatives," MPR Technical Report DSS-RA-890120, Microtel Pacific Research Limited, Burnaby, B.C.

[GRAY81] Gray, J., McJones, P. et al, "The Recovery Manager of the System R Data Base Manager," Computing Surveys, Vol. 13, No. 2, June 1981

[KAME83] Kameda, T. "Distributed Systems Seminar", lecture notes, Simon Fraser University, Burnaby, B.C. 1983.

A Transaction Model for Online Transaction Processing Systems

Dr. Bill Claybrook
Stratus Computers, Inc.
55 Fairbanks Blvd.
Marlboro, MA 01752

1.0 Introduction

This paper describes the characteristics of online transaction processing (OLTP) systems and online transaction applications (OLTAs). A transaction model is described that handles multi-commit and distributed transactions using the concept of a transaction control unit (TCU). The model is shown to be sufficient for modeling and implementing commercial OLTP systems.

2.0 Characteristics of OLTP Systems and Online Transaction Applications

With online transaction processing (OLTP), many users (operators) are typically using, in an interactive fashion, an online transaction application (OLTA). For online processing, input data are not normally prepared before processing begins; instead, it is entered interactively as needed, usually from many different users working concurrently. Within an OLTA environment, several users can potentially be updating a single file or table simultaneously. When a failure occurs, the application cannot simply be rerun or continued (from the point of failure) because the state of processing is unknown. OLTAs require a system that provides special mechanisms for recovery and restart.

When OLTAs are designed, internal memory, processor time, and external storage are all resources that should be considered. However, unlike batch processing, there are additional resources that should be considered, namely, response time, exclusive use of shared resources, e.g., records and files, and communication line transmission capacity and speed. With OLTAs, the terminals (workstations) may be located long distances from the host machine and information is usually carried over public telephone lines.

A conceptual model of an OLTP system is described below in order to provide a baseline for discussion in the remainder of this paper. This model can be used to describe the interaction and functionality of the various pieces of software that comprise an OLTP system. The model does not necessarily reflect the architecture of any commercial system, but commercial systems can be describe using the attributes of the model.

Due to limited space, we only list the components of the model and then briefly describe some of the more important interactions. The components include:

1. Transaction processing user (TP user).

2. Terminal or workstation.

3. Form - interface for TP user.

4. Menu - special form used to select work within an OLTA.

5. Transaction processing routine (TPR) - routine (or step) within a transaction that performs some type of access to a database. In some OLTP systems, a TPR is executed by a server process.

6. Transaction - one or more TP user input messages.

7. Application configuration - a specification (in some language) of the environment in which transactions execute.

8. Configuration specification facility - a language or menu-driven interface used to specify an OLTA environment.

9. Forms management system - a utility used to define forms for the OLTA interface.

10. OLTA - a collection of terminal control program(s), forms, menus, TP runtime services, TPRs, and an environment specification.

11. Transaction processing monitor (TPM) - a program that monitors the interaction between terminals and the OLTA and the rest of the OLTP system.

12. Transaction control program (TCP) - program that handles the interaction between users and the OLTA.

13. Process group - processes designed to share resources such as files and database subschemas.

A TCP receives a transaction-invoking message from a user, sends a message to a TPR to perform a database action, and sends a message to the user on behalf of the TPR. A TPM may perform some or all of the following functions: manage login/logout and security, schedule server processes to execute TPRs, monitor and control the execution of transactions, e.g., multi-commit and distributed, manage message queues, perform concurrency control and recovery functions, manage communication over a network to remote nodes in a distributed system, etc.

An OLTA is capable of executing several types of transactions. In a requester - server system these transactions are defined as entry points in a TCP. A TP user chooses the transaction to execute via a transaction selection menu provided as part of the OLTA. For example, one transaction might insert a row in a table, another might schedule a meeting, etc.

Specifying the configuration of an OLTA involves specifying some or all of the following: files and database schemas to be used by transactions, set of transactions defined for the application, maximum number of terminals (workstations), both local and remote that can be attached to the OLTA, maximum number of transactions that can be active at any given time, maximum message length, number of message buffers to be allocated for holding messages, login names of TP users, identifiers of the terminals that can attach to the OLTA, names of forms and menus, limits on elapsed time per transaction and the total inactivity allowed at a terminal, number of

TCPs, number of processes to execute TPRs, and the state to return terminals to during recovery after a system crash.

Various architectures have been used to build OLTP systems and hence configure OLTAs. These include single process- single threaded TPM (one process per user), single process - multi-threaded TPM (CICS-like), and client (requester) - server architectures. Most commercial OLTP systems can be mapped to one of these three architectures. With the client - server architecture, the client process may be running on the same machine as the server processes or the client process may be running on a workstation (including PCs) connected to the server machine via a communications protocol over ethernet or over a long haul network.

The one process per user architecture is relatively easy to engineer using a time-sharing operating system. It has the reported disadvantages that performance may be poor since there are many processes and much process context switching. In addition, large amounts of memory may be used since each bound unit contains all of the OLTA code. The one process - multi-threaded TPM architecture has one process for all terminals, and it switches among the terminals when they are ready to enter a message. The process implements multi-tasking (or threads). The assumption here is that thread switching (in user space) is more efficient than process context switching. The disadvantages are that the TPM is large and complex and all access for files is done through the TPM.

The client (requester) - server architecture is considered to be the most appropriate architecture for developing OLTAs. The reason for this is that the client process may be running on a workstation or PC executing an application program that is making requests to a server machine that provides access to databases. The first use of this architecture and still a popular way of utilizing it, however, is to have requester and server processes running on a single machine. The requesters accept data from a terminal and send requests to server processes. Developing an OLTA usually means writing requester and server programs. However, most DBMSs have client products that allow users to send requests to servers by writing little, if any, code. It should be noted that there are some significant differences between clients running on a machine separate from the server machine and the client and servers running on the same machine. For example, requester processes are multi-tasked in the more traditional implementations of requester-server systems.

Whereas in the other implementation where clients originate on PCs, there is one client process per user. Other differences will become evident in the remainder of this paper.

3.0 The Transaction Model

The transaction model described in this section is appropriate for all resources in a system not just files and databases. One approach to placing other system resources under transaction control is to utilize a uniform I/O system interface. Exactly, how useful this is in a real OLTP system is somewhat questionable. However, we are open to arguments favoring it.

A *transaction control unit* (TCU) provides the transaction control for a class of resources. A TCU has a layered architecture with a transaction manager (TM) at the highest layer, a commit manager (CM) at the next lower layer, and a resource manager (RSM) at the lowest layer. A TCU manages resources in its environment. The resources managed in one TCU are not necessarily like the resources in another TCU. Typical TCUs are file system managers and DBMSs. Each node in a network has a top level TCU, designated by TCU_1. TCU_1 manages all TP resources on the node. These resources are processes, messages, shared memory, etc. All remote accesses to a node are handled by TCU_1 on that node.

The architecture of a TCU is illustrated in Table 1 by listing the interface to each layer from left to right. The table should be interpreted in the following manner. A transaction issues a *TM_write* operation, the *TM_write* executes a *CM_write* operation, and the *CM_write* issues a *RSM_write* operation.

Each class of resources is managed by a RSM. RSMs maintain the *consistency* and *security* of resources, but they do not maintain the *integrity* of resources. A typical RSM is a file manager that manages all files in a file system. Multiple file systems may exist with each having its own RSM. For example, UNIX System V.3 permits multiple file systems to exist simultaneously. A CM is responsible for maintaining the integrity of the resources managed by its RSM partner. A CM handles all commit, rollback (abort), and recovery functions for the resources. In addition, a CM implements a "prepare to commit" operation for participating in the two phase commit protocol.

A TM is responsible for coordinating the efforts of its associated RSM and CM. One of its primary functions is to manage the execution of multi-commit transactions, i.e., transactions that involve two or more TCUs, and distributed transactions. A distributed transaction is defined within the context of a single TCU. For example, a transaction that involves remote accesses within the context of a file system is a distributed transaction. A multi-commit transaction may involve TCUs residing on remote nodes so it is distributed in that sense.

Table 1
Architecture of a TCU

TM	CM	RSM
start_transaction		
abort_transaction	abort_transaction	
commit_transaction	commit_transaction	
	prepare_transaction	
read	read	read
write	write	write
open	open	open
close	close	close
		create
		destroy
		grant
		revoke
connect		connect
disconnect		disconnect
send		send
receive		receive

A TM typically maintains the status of each transaction and allocates and manages a transaction descriptor for each transaction. A transaction descriptor is a repository of information for each transaction. For example, a transaction within a single TCU can be distributed over one or more remote nodes, and the identifiers of the nodes can be maintained in the transaction descriptor. The TM executes the 2PC protocol when its TM_commit_transaction operation is executed. The algorithms used in the TM, CM, and RSM for one TCU may be significantly different than those used for the TM, CM, and RSM in another TCU.

The execution of a transaction may result in a hierarchy of TCUs being invoked (see Figure 1). TCUs subordinate to TCU_1 are labelled TCU_{11}, TCU_{12}, etc. If each TCU in the hierarchy is replaced by the transaction executed in its scope, then we have a hierarchy of transactions with the transaction at each node in the hierarchy being the parent of the subordinate transactions at the next level. Subordinate transactions are subtransactions (of their parent). This leads naturally to nested transactions. The transaction at the top of the hierarchy is referred to as the top level transaction (TLT). A TCU at any level can be designed to handle multi-commit transactions occurring at that level. This means that multi-commit transactions can originate within the scope of any TCU unless the OLTP system is not engineered to provide this general capability.

Insert Figure 1

3.1 Examples

Two examples are presented and discussed to illustrate some of the concepts embodied in the transaction model. These examples illustrate the simplicity and utility of the model. In the examples, the "name" of the TCU is prefixed to all operations. The absence of a name implies TCU_1.

Example 1

```
sql_start_transaction
        sql_read
        sql_write
        sql_write
        sql_read
sql_commit_transaction
```

Example 2

```
start_transaction
        fileio_read
        sql_write
        sql_write
        fileio_write
        db2_write
        db2_read
        sql_write
commit_transaction
```

In Example 1, only the sql TCU is involved. The transaction may be distributed. If so, sql_TM keeps a record of all remote nodes accessed and then executes the 2PC protocol when the sql_commit_transaction statement is executed. This involves issuing a sql_CM_prepare operation to the sql_TCU on each remote node involved in the transaction.

The transaction in Example 2 is obviously a multi-commit transaction. Since the subordinate TCUs are not explicitly specified via, for example, fileio_start_transaction, sql_start_transaction, and db2_start_transaction statements, there is a question as to what actually happens during transaction execution. We will alter the transaction by giving the start_transaction some parameters, namely the names of the subordinate TCUs (this can also be done for multi-commit transactions executed by subordinate TCUs), i.e., start_transaction(fileio, sql, db2). This is not the only solution to the problem, but it is probably the simplest at the expense of requiring the user to do a little extra work.

Now when start_transaction(fileio, sql, db2) is executed, fileio_start_transaction, sql_start_transaction, and db2_start_transaction operations are executed. We assume that the two fileio operations form one subtransaction, the three sql operations form a second subtransaction, and the two db2 operations form third subtransaction. Four transaction identifiers (TIDs) are created--one for the TLT and one each for each of the three subtransactions. Each transaction has a transaction descriptor created for it by the TM in its TCU. If any of the subtransactions are distributed, then the TM in their TCU executes the 2PC protocol. TCU_1 saves the name of each TCU invoked within its scope, possibly in the transaction descriptor created by TM_1. When commit_transaction is executed, it issues fileio_CM_prepare, sql_CM_prepare, and db2_CM_prepare operations. TM_1 collects the votes on whether or not to commit the transaction. It commits the TLT if they all vote to commit or aborts it if one or more cannot commit.

The transaction in Example 2 has three implicitly defined transactions nested inside it. The subtransactions could be explicitly defined by using explicit start and commit operations to define the scope of each subtransaction. For example, the fileio subtransaction would appear as

```
fileio_start_transaction
        read
        write
fileio_commit_transaction
```

In the fileio transaction, the name of the TCU must be placed in the TLT's transaction descriptor when the fileio_start_transaction operation is processed. When fileio_commit_transaction is executed, the subtransaction should not actually commit, but it should prepare to commit instead. Perhaps the fileio_prepare_transaction operation should be explicitly used instead of the fileio_commit_transaction operation.

It should be pointed out that the semantics of the transaction may be altered if Example 2 is changed to use explicit subtransactions. Another alternative is to treat each operation as a subtransaction; however, there is more overhead associated with this approach.

4.0 Utilizing the Transaction Model

In this section we show that the transaction model is sufficient for describing the actions of existing OLTP systems. We look at the Stratus requester - server architecture with TPF (transaction processing facility) and the Oracle 6.0 with TPS (transaction processing subsystem) DBMS

4.1 Stratus Requester - Server Architecture with TPF

The requester processes are multi-tasked with the switching among tasks controlled by a monitor task (one per requester). Requester tasks accept input primarily from terminals. Transactions are defined in requester programs. TPF is a no undo/redo transaction processing system with a transaction processing overseer (TPO) process that performs all commit processing.

Presently, we are working on a TP monitor that will start requester and server processes at OLTA startup and a common commit manager that will allow transactions containing TPF subtransactions and SQL/2000 subtransactions to be executed within a top level transaction. The TP monitor will also perform automatic load balancing by starting additional servers when necessary.

The TP monitor and the common commit manager can be modelled using the transaction model described in Section 3.0. RSM_1 would manage the creation of requester and server processes during OLTA startup and handle dynamic load balancing during transaction processing. The common commit manager would be implemented through the cooperation of TM_1 and CM_1. Thus, TCU_1 would essentially provide the functionality of the TP monitor and the common commit manager. TPF and SQL/2000 would be treated as subordinate TCUs, say TCU_{11} and TCU_{12}.

4.2 Oracle TPS

The TPS implementation of SQL*Net is a generalization of the Oracle 5.0 two task concept that implements the two-task concept across networks. For two tasking there is a user process - Oracle server process pair. This permits a client (user) process to run on a PC or workstation and the server process to run on another machine that acts as the server. A client process issues a CONNECT operation to make a physical connection with a remote server. As a result of this connection, the server machine creates an Oracle server process that is "connected" to the client process. The client process effectively creates the server process. There are also a number of background processes (on a per database basis) that perform logging, writing data blocks to disk, etc. These background processes are started via a startup command issued by SQL*DBA.

The TPS two-task implementation described above can be modelled using the TCU concept. For example, TCU_1 (actually its components) will handle the CONNECT operation issued by a client. RSM_1 will listen for CONNECTs and accept client connection requests. RSM_1 will start Oracle server processes; thus, the client and server will know each other only through the message-based system of SQL*Net instead of the client being responsible for starting the server process. RSM_1 will start all Oracle background processes, and if a background dies RSM_1 will terminate the corresponding database's Oracle server processes.

RSM_1 can also prime the system with Oracle server processes so that a CONNECT from a client can be done more quickly. The client process and the Oracle server process communicate via a two-way direct queue created by RSM_1 and opened first by the Oracle server process and then by the client process. The TPS kernel running on the server machine is a subordinate TCU.

5.0 Summary

This paper has described a transaction model that can be utilized in OLTP systems for handling multi-commit and distributed transactions. Its generality permits it to model the functionality normally found in TP monitors as well as the functionality found in file systems and DBMSs.

Figures

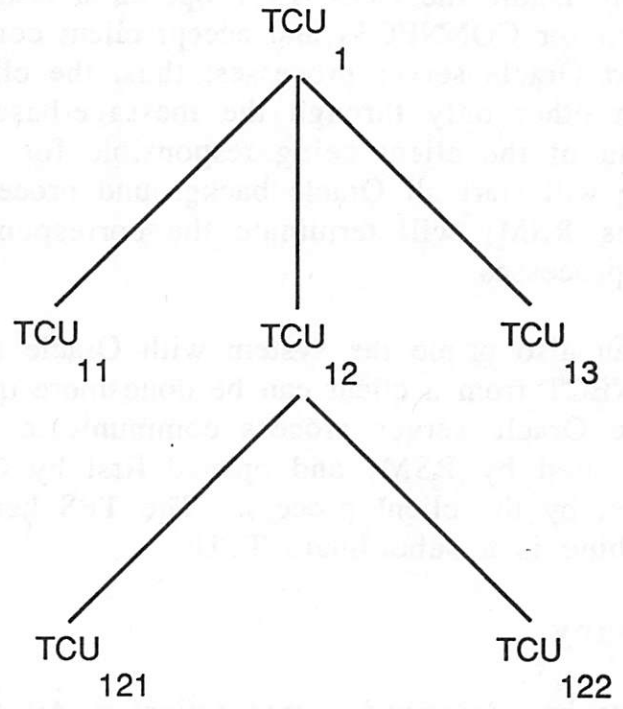


Figure 1. Hierarchical nature of TCUs

Protocols for Distributed and Nested Transactions

Dan Duchamp
Computer Science Department
Carnegie-Mellon University[1]

Abstract

This paper presents three new results in transaction management protocols: two sophisticated new protocols for management of distributed transactions (a so-called "non-blocking" commitment protocol and an abort protocol for transactions that may be nested), and an improved version of the venerable two-phase commit protocol.

This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520.

---

[1]Author's current address: Computer Science Department, Columbia University, New York, NY 10027. The former Computer Science Department at Carnegie-Mellon University is now called the School of Computer Science.

One of the opportunities presented by having UNIX-based transaction processing facilities is that, because of the popularity of UNIX, transaction systems may soon execute a wider variety of workloads in a wider variety of environments. In other words, transaction processing will become a *technology* rather than simply a specialized field. If this is true, the model of what a transaction is will likely become richer, and there will be a need for more sophisticated transaction management protocols.

This paper presents three new results: two sophisticated new protocols for management of distributed transactions (a so-called "non-blocking" commitment protocol and an abort protocol for transactions that may be nested), and an improved version of the venerable two-phase commit protocol. The two new protocols are both too complicated to be fully explained in the limited space here; full coverage can be found in [3] and [4], respectively.

1. Non-Blocking Commit

Two-phase commitment is an excellent protocol: it is simple and fast. However, it has one significant drawback. For a period of time all of the information needed to make the decision about whether to commit or abort — that is, whether or not all subordinate sites are prepared — is located at the coordinator and nowhere else. If, during this period of time (called the ***window of vulnerability***), a subordinate loses contact with the coordinator (either because the coordinator crashes or because a network failure prevents the two sites from communicating), then the subordinate must remain prepared until the failure is repaired and communication with the coordinator is reestablished. Until then, the subordinate continues to hold its locks for the transaction, and is said to be ***blocked***.

The new three-phase "non-blocking" commitment described here continues to operate in spite of any single site crash or network partition. The protocol is correct despite the occurrence of any number of failures, although it may block if there are two or more failures. It is impossible to do better [10, p. 83]. When there is no failure, the protocol requires three phases of message exchange between the coordinator and the subordinates and requires each site to force two log records. Read-only transactions are optimized so that a read-only subordinate typically writes no log records and exchanges only one round of messages with the coordinator, just as with two-phase commit.

The protocol makes five changes to two-phase commit:

1. The prepare message contains the list of sites involved in the transaction, as well as some additional information specifying the degree of replication needed in the "replication phase" explained in item 3 below.
2. The subordinates do not wait forever for the commit/abort notice, but instead timeout and become coordinators. The transaction can be finished by a new coordinator that was once a subordinate. Simultaneous action by several coordinators is possible, but is not a problem.
3. An extra phase (called the ***replication phase***) exists between the two standard phases. During the replication phase, the coordinator collects the information that it will use to make the commit/abort decision, and replicates it at some number of subordinates. The subordinates write the information in stable storage, just as they write a prepare record and a commit record. The coordinator is not allowed to make the decision until the information has been sufficiently widely replicated to exclude the other outcome. This is the well-known quorum consensus method [5].

4. No site ***forgets*** (i.e., expunges its data structures) about a transaction until all sites have committed or aborted.
5. The coordinator prepares before sending the prepare message.

The first two changes are quite intuitive. The first simply gives subordinates the ability to communicate after the loss of the coordinator. The second tells how they communicate: by having a subordinate become a coordinator and tolerating the presence of multiple coordinators, there is no need for electing a new coordinator. The third change prevents a partition from causing incorrect operation by ensuring no site can commit or abort until it is certain the other outcome is excluded.

The full protocol is quite complicated (it is described in [3]), but can be summarized:

- *Prepare phase*: coordinator gives subordinates the list of sites and the quorum sizes, and asks them to become prepared.
- *Replication phase*: coordinator tells subordinates whether all sites are prepared or not.
- *Notify phase*: coordinator tells subordinates the outcome.
- *Forget message*: coordinator tells subordinates to forget.

The commitment point is reached during the replication phase once a commit quorum of sites have recorded that all sites are prepared; then, commit is inevitable.

2. Abort for Nested Distributed Transactions

Designing an abort mechanism for a system that supports Moss-nodel nested transactions [9] is quite complicated, for several reasons. With nested transactions, aborting a single transaction may require aborting many others (specifically, its descendants), and aborting these transactions should not prevent the remaining parts of the family from committing. The transactions to be aborted may still be spreading to new sites and/or creating new nested transactions while abort is proceeding. Also, the information recording the extent of spreading and nesting is only in memory, and therefore is lost if a site crashes. There are two consequences of losing this memory.

First, a crash potentially produces orphans. Co-existing with orphans there may be other transactions within the same family that should commit. The orphaned and non-orphaned work must be distinguished. Unfortunately, the messages of the commit protocol identify only the family and mean "commit all the operations of family X." Consequently, if orphaned work is not detected by commit time, it will be committed. A system without nested transactions could afford to be slower in cleaning up orphans because the presence of orphans would not threaten the atomicity of any transaction.

Second, because any site may crash at any time and thereby lose all memory of the transactions that are or will be aborted, an abort protocol "cannot depend upon anything." For instance, it cannot wait for acknowledgement that a particular transaction have been undone, since it may be impossible for such an acknowledgement ever to be sent. Consequently, an "abort mechanism" really consists of two portions:

1. An "abort protocol" which is a method for locating and undoing as much as possible.
2. An "orphan protection mechanism" for ensuring the orphans that escape the abort protocol (due to failures) never commit.

If the abort protocol cannot undo everything, it is responsible for ensuring that the orphan protection mechanism has enough information to do its job.

In addition to the inherent difficulties of aborting, there are several more practical restrictions. First, a transaction cannot commit until all of its children are either committed or aborted. Second, it is unacceptable to add extra information to messages sent between clients and servers — as Argus does [7] — in order to facilitate orphan detection. Third, all inter-site abort protocol communication should be done with datagrams, since connections cannot be depended upon because connection failure may be the very cause of the abort. Last, recovery requires that whenever a transaction aborts, it and all its active or committed descendants (but no descendants that have already aborted) must be identified together in a single call to the recovery manager. In other words, the recovery algorithm can undo any particular nested transaction only once.

When abort is requested for a nested distributed transaction, the abort protocol travels from site to site, undoing all the operations of all the transactions that must abort. If an attempt to reach some site fails (that is, the site is a "dead end"), then the creation site of the top-level transaction (the ***commit source***) is given the identity of the dead end site and the protocol terminates having accomplished only partial abort. A dead end that is reported to the commit source is called a ***dangerous site***. In this fashion, aborting a nested transaction may cause the commit source to accumulate a list of dangerous sites. The list is the link between the abort protocol and the orphan protection mechanism. A dead end is dangerous for two reasons:

- If the dead end is crashed, and if a call was outstanding to it, the transaction being aborted may have created descendants which may have spread (or still be spreading) to other unknown sites. These calls are orphans that must not be allowed to commit.
- If the dead end is unreachable due to communication failure, orphaned work will be left there, possibly together with work that should commit.

Any orphaned work will either be active at a dangerous site, or will have passed through a dangerous site on its way to its destination.

The orphan protection mechanism is built into the commitment protocol. When the commit source begins the commitment protocol, it will list all dangerous sites in the prepare message. Subordinates will process such a "dangerous prepare" message somewhat differently from a typical prepare message. A subordinate transaction manager will get from its communication manager its list of accumulated sites for the family. The list will include every site used to service any request for which the subordinate was a destination or a source. If any dangerous site is in the list, the subordinate must vote to abort the entire (top-level) transaction. This approach conservatively assumes that any correspondence with a dangerous site is orphaned work, and avoids committing it by voting to abort the entire transaction. This rule overestimates the amount of orphaned work (any work that used any dangerous site), and overaborts (the whole family).

The key to the correctness of this method is informing the commit source when a dangerous site is detected. The restriction that a family cannot be committed until all of its children are either committed or aborted is used to ensure that orphans do not commit: the synchronous abort call does not return until the possibility of orphan problems is recorded at the commit source, thereby delaying the commitment of the enclosing transaction. This feature is quite unfortunate, but is required in order to safely abort distributed nested transactions without burdening client-server messages with complex orphan detection information, as in Argus [7]. If the commit source is unreachable (because of crash or partition), then the top-level transaction is aborted. Therefore, the protocol continues to operate in spite of any number of failures; "operating" may include aborting the top-level transaction if the abort of a nested transaction becomes blocked. Aborting a top-level transaction is a unilateral operation that does not block.

The steps involved in aborting an arbitrary nested transaction are:

- Discover which transactions are ***victims*** (i.e., which transactions must be aborted).
- Go from site to site undoing victims.
- If a dangerous site is encountered, record it at the commit source before returning to the caller.
- If recording danger fails, abort the whole family.
- Return to the caller once either all victims are aborted, or all dangerous sites are recorded at the commit source.
- If dangerous sites were found during abort, list them in the prepare message of the commit protocol.

The goals of an abort mechanism are, in order of importance:

1. *Atomicity*: Ensure that no site commits and aborts the same transaction.
2. *Efficiency*: Do not add to the overhead of normal processing only to facilitate aborting. Specifically,
   a. Do not add log forces.
   b. Do not add messages to the commit protocol.
   c. Do not make any inter-TranMan message be proportional in size to the number of transactions executed within a family, since this number could be very large.
   d. Do not add server-visible information to client-server RPCs.
3. *Flexibility*: Allow ***unilateral abort***, which means that unless a transaction is prepared, a site may abort the transaction without having first to communicate with any other site.
4. *Speed*:
   a. Drop locks as fast as possible.
   b. Do not delay the commitment of an enclosing transaction beyond the return of the abort call of the nested transaction.
   c. Resume the process that (synchronously) invoked abort as soon as possible.
5. *Cleanliness*: Exterminate ***orphans*** as quickly as possible. An orphan is any operation, finished or still being performed, that must be aborted but which cannot be located. Orphans are created by crashes because the record of which sites call which other sites is kept in memory, not stable storage. For example, if Site A invokes an operation at Site B and then Site A crashes, then the work done at Site B is an orphan. It does not matter whether the operation replies before the crash.
6. *Resilience*: To the greatest extent possible, continue operating in spite of failures. (Abort often happens because of failure.)
7. *Liveness*: When aborting a nested transaction, to the greatest extent possible do

not unnecessarily abort enclosing transactions.

If the transaction being aborted is top-level, then some of these goals (such as 4b and 7) degenerate. Unilateral abort is a somewhat abstruse but traditional goal [6]. Its motivation is that a command to abort a transaction may represent an "emergency" at the aborting site, and that any unnecessary delays in undoing the transaction at that site are not tolerable. The abort mechanism makes an estimate of the possibility of orphans; sometimes, because of lack of information, this estimate is an overestimate. The liveness goal is intended to ensure that there are few unnecessary overestimates.

The one-phase abort protocol and associated orphan protection method completely satisfy all of the goals except these, which are mostly, but not completely, satisfied:

3. *Flexibility:* Distributed nested transactions cannot abort unilaterally, although top-level transactions can.
4. *Speed:*
    c. The abort call is synchronous; that is, the caller does not regain control until the protocol has finished.
5. *Cleanliness*: The only guarantee made about orphans is that they will never commit. They can continue to compute.
7. *Liveness*: The list of dangerous sites represent an *estimate* about the extent of the existence of orphaned work. This estimate may sometimes be too great; it is never too small.

The inability to unilaterally abort an arbitrary Moss-model nested transaction is fundamental. However, failing to meet goals 5 and 7 is the result of meeting the efficiency goal. By not placing extra information in stable storage and in client-server messages, it becomes impossible to distinguish whether a given operation is an orphan (without consulting the transaction's creation site). Consequently, this abort mechanism does not provide "strong" orphan protection which detects an orphaned computation upon first contact with it, but rather makes an overestimate of the extent of existence of orphans at commit time. Therefore, failure to meet goal 4c is a consequence of not having strong orphan protection: the abort call cannot return until the information needed to make the orphan estimate is safely stored at the commit source.

Because abort is assumed to be rare, to an extent its performance is not one of the important parameters of a transaction facility. It is far more important that the abort mechanism intrude as little as possible on the typical behavior of the system (i.e., failure-free commitment). In this sense, the abort mechanism discussed here succeeds: all processing related to aborting a particular transaction happens only after the abort call has been invoked. The other important performance goal is to continue operating in spite of failures. The resilience goal is met, but at a cost: it is never necessary for the abort protocol to wait for a message from another site, but the top-level transaction may have to be aborted in order achieve this.

3. Performance Improvement to Two-Phase Commit

The basic two-phase distributed commitment protocol as described in [8, pp. 381-382] can be optimized so that a subordinate update site drops its locks more promptly and makes one fewer log force per transaction. The optimization applies as well to the variations of two-phase commitment (i.e., hierarchical, presumed commit, and presumed abort) described in the same paper.

As described in [8], the sequence of actions during the second phase of an execution of two-phase commitment in which the transaction commits is:

1. Coordinator forces commit record into its log.
2. Coordinator drops its locks.
3. Coordinator sends commit notice to subordinate.
4. Subordinate *forces* commit record into its log.
5. Subordinate drops its locks.
6. Subordinate sends commit acknowledgement to coordinator.
7. Subordinate forgets about transaction.
8. Upon receipt of all commit acknowledgements, coordinator forgets about transaction.

An optimization is possible wherein events 4 through 6 become:

4. Subordinate drops its locks.
5. Subordinate *spools* commit record into its log.
6. Subordinate sends commit acknowledgement to coordinator once the spooled commit record has been placed in the log.

The subordinate drops its locks *before* writing a commit record.

Of course, a subordinate may not drop its locks until the transaction is committed. In the unoptimized protocol, a subordinate writes its own commit record to indicate that the transaction is committed and therefore that locks may be dropped. The optimized protocol uses the commit record *at the coordinator* to indicate the same fact. So the coordinator must not forget about the transaction before the subordinate writes its own commit record; hence, the commit acknowledgement cannot be sent until the subordinate's commit record is written.

The advantages of the optimization are:

1. Throughput at the subordinate is improved because fewer log forces are required. The amount of improvement is dependent upon the fraction of transactions that require distributed commitment.

2. Locks are retained at the subordinate for a slightly shorter time; this factor is important only if the transaction is very short.

Throughput is improved at no cost to latency. This optimization is independent from the notion of group commit [1, p. 7], which is a more widely applicable technique that improves throughput at the cost of increasing latency.

The disadvantages of the optimization are:

1. Some mechanism must exist to delay the sending of the commit acknowledgement notice until after the subordinate has written its commit record. (A system that implements group commit very likely already has this mechanism.)

2. The subordinate's "window of vulnerability" is increased.[2]

Both disadvantages are relatively minor, and are worth suffering considering that the optimization can result in noticeable throughput improvement, as reported in [2].

An important observation is that, even using the optimization, the serialization order of two transactions remains the same even if the second obtains the locks dropped "early" at a subordinate by a prior transaction whose commit record is not yet logged. That is, the contents of

[2]The window is the time between the moment when the subordinate writes its prepare record and the moment it writes its commit/abort record. If the subordinate crashes during this window its recovery is dependent upon the availability of the coordinator.

the subordinate log are the same whether or not the commit record of the first transaction is forced before its locks were dropped. This is true provided that the first log record written by the second transaction is either in the same commit group as the commit record of the first (for systems implementing group commit) or is always one that is forced, not spooled (for systems not implementing group commit). The simple case analysis below shows that the serialization order is preserved in systems not having group commit; the argument for systems with group commit is trivial.

Suppose that transaction 1 has committed and dropped its locks at a subordinate site X, but that its commit record has not yet been placed in the subordinate log. Suppose further that transaction 2 then locks some of the data that was updated by transaction 1. The possible next actions and their consequences are:

- Site X crashes: the recovery process aborts transaction 2 and re-prepares transaction 1. Since the commit acknowledgement for transaction 1 has not yet been sent, the subordinate queries the coordinator and discovers that transaction 1 is committed.
- Transaction 2 aborts: same as if transaction 2 never executed.
- Transaction 2 commits:
    - Transaction 2 is read-only: same as if transaction 2 never executed.
    - Transaction 2 is not distributed: the commit record of transaction 2 is forced, thereby placing the commit record of the first transaction in the log before it.
    - Transaction 2 is distributed, and Site X is the coordinator: the commit record of the second transaction is forced.
    - Transaction 2 is distributed, and Site X is a subordinate: the prepare record of the second transaction is forced.

References

[1] D. J. DeWitt, et. al.
Implementation Techniques for Main Memory Databases.
In *Proc. ACM-SIGMOD 1984 Intl. Conf. om Mgmt. of Data*, pages 1-8. May, 1984.

[2] D. Duchamp.
Transaction Management.
PhD thesis, Carnegie-Mellon University, 1988.
Available as Technical Report CMU-CS-88-192, Carnegie-Mellon University.

[3] D. Duchamp.
A Non-blocking Commitment Protocol.
1989.
submitted to ACM Trans. on Database Systems.

[4] D. Duchamp.
An Abort Protocol for Nested Distributed Transactions.
1989.
submitted to ACM Trans. on Database Systems.

[5] D. K. Gifford.
Weighted Voting for Replicated Data.
In *Proc. of the Seventh Symp. on Operating System Principles*, pages 150-162. ACM, December, 1979.

[6] B. Lindsay et. al.
Computation and Communication in R*: A Distributed Database Manager.
ACM Trans. on Computer Systems 2(1):24-38, February, 1984.

[7] B. Liskov, R. Scheifler, E. Walker, and W. Weihl.
Orphan Detection.
Technical Report Programming Methodology Group Memo 53, MIT, February, 1987.

[8] C. Mohan, B. Lindsay.
Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions.
In *Proc. Second Ann. Symp. on Principles of Distributed Computing*, pages 76-88. ACM, August, 1983.

[9] J. E. B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, 1985.

[10] D. Skeen.
Crash Recovery in a Distributed Database System.
PhD thesis, Univ. of California, Berkeley, May, 1982.

Distributed Transaction Integrity

Dorm Fulton
Unisys, Roseville
P. O. Box 64942 - WE2C
St. Paul, MN 55164
uunet!s5000!dlf

ABSTRACT

This paper will consider a transaction to be a unit of work which exhibits the ACID properties. The distributed transaction paradigm is that of cooperating transactions executing on multiple distributed hosts. Distributed transaction integrity is based upon two capabilities. The first is the ability of a given transaction system to recover after a failure. The second is the ability to coordinate the recovery across multiple transaction systems. In exploring distributed transaction recovery mechanisms, it is necessary to review some data base recovery scenarios for local transactions. This review will maintain an awareness of distributed recovery synchronization implications and will show that advantages can be gained through the use of common recovery schemes. Integrity in the distributed transaction environment will normally require a two phase commit protocol. Variations on this protocol provide for additional responses.

1. Introduction

This paper will address integrity across a network of cooperating transactions which comprise a distributed transaction. This paper will not attempt to explore the record or file locking that might be desirable in these environments. The assumption regarding locks is that records that are to be updated will be exclusively held from the read for update until the distributed transaction is complete.

A transaction is defined as a logical unit of work which exhibits the ACID properties of atomicity, consistency, isolation, and durability. Atomicity is the attribute of a transaction which insures that either all changes and updates resulting from a transaction's execution are made permanent, or all are nullified, that is, backed out. Consistency means that a successful transaction transforms a data base from the previous valid state to a new valid state. Isolation is the transaction attribute that insures the effects of the transaction's execution do not become visible to other transactions until this transaction commits. Durability means the committed data base updates will survive subsequent system or media failures.

Transactions manifest themselves as logical units of work which are defined by an application program initiating transaction start, usually as the result of a triggering event. Transactions may span multiple application programs, data base managers and transaction systems. The atomic quality of transactions occur between start and end brackets which define the period in transaction mode. In general transactions execute within specific environments under the control of a transaction manager or monitor. These environments are generally oriented to a specific aspect of the enterprise, which represents a collection of applications programs and a data base. We will refer to these application specific groupings as a transaction system with the understanding that transaction systems may incorporate multiple data base managers.

This document will consider two levels of transactions. The first level consists of local transactions inherent to data base managers. Local transactions may also be evident within transaction systems for cases where the data base manager does not maintain a concept of transactions. These local transactions maintain the ACID principles at their local level. The second level is that of the distributed transactions which are spread across multiple transaction systems in a network. These transactions are referred to as global transactions in other documents. Distributed transactions are mapped to one or more local transaction providing brackets which can incorporate multiple local transactions into the atomicity of the distributed transaction.

The distributed transaction paradigm normally incorporates multiple local transactions cooperating across a network of distributed transaction systems. Variations on the paradigm include multiple transaction systems on the same host where intra-host communications between the transaction systems is analogous to the inter-host communications of the physically distributed model. The synchronization problems of distributed transactions also occur with multiple data base managers in a single system. This case provides multiple local transactions within the single transaction system and necessitates coordination inherent with distributed transactions.

Normal transaction termination causes the effects of the transaction, including data base updates, to be made permanent. The term used to reflect this normal completion is commit. Commit of a distributed transaction causes the changes which result from each local transaction to also be made permanent. Abnormal termination of a transaction causes a back out of the effects of the transaction, nullifying the changes which occurred as a result of the transaction's execution. The transaction abort results in rollback of any data base updates. In most cases the abort of a local transaction will result in the abort of the distributed transaction to which it is a participant.

2. Transaction System Recovery

Three levels of recovery can be provided by transaction systems to cover the possible failures which may occur. These are the rollback and restart of the individual transactions, the restart of the transaction system, and data base restoration. Individual transaction rollback is the basic level necessary to maintain data base consistency. The transaction system or data base manager must be capable of nullifying the effects of an individual transaction when it fails. The methods used to achieve this rollback are discussed under Data Base Recovery. The optional restart of the failed transaction from the retained input request or message will essentially replay the transaction after rollback. This is referred to as message recovery and is provided on some systems. Transaction system restart is accomplished after system restart by rollback of the active individual transactions. This will bring the data base to the consistent state prior to the execution of the transactions that were active at the time of failure. Further recovery can be accomplished if the system supports message recovery and can now replay the active transactions from their input request. Restoration of the data base is provided to recover from a failure of mass storage where the data base resides. This methodology is also covered in the next section. It should be pointed out that data base restoration is intended to be followed by system restart.

In order to insure data base recovery and restoration it is necessary to secure certain data. This is often referred to as logging. Generally no consideration is given to the transient nature or temporary need for some of the information. Typically the information necessary to complete or rollback a specific transaction is only critical during the lifetime of the transaction in question. Once the transaction has committed this information is of no further value. Whereas, the information needed to restore the data base in the event of failed mass storage devices is both minimal and permanent until the next data base archive. We will refer to the temporary data container as the retention file, and the longer term restoration data as the log.

3. Data Base Recovery

Data base integrity has two general problems to overcome. The first is insuring that the changes which occur as a result of a transaction can be made permanent or backed out. The second is insuring the data base can be restored in the event of a mass storage failure.

Two methods are used to insure the ability to complete an update or nullify the changes made during the course of the transaction. These methods exhibit the same basic concepts but vary essentially in the timing of the actual update to the data base. The methods also differ in the information which is needed to guarantee the update can be made or back out can occur. That is, the records kept to insure atomicity and consistency differ as a result of this timing difference. This paper will refer to these two methods as direct update and deferred update, with the names reflecting the difference in update timing. This timing difference requires that processing during commit or abort be unique to each method.

The direct update method updates the data base at the point where the application program executes the update request. This method provides a scenario where the updating is done prior to commit with the provision to rollback if the transaction aborts. The data base updates are synchronous with the application program's execution. The actual modification of the date base will result with the command to update a data image. This method requires that a copy of the original unmodified data record be retained such that the updates can be nullified or rolled back if the transaction does not finish normally. The saved original data records are called either before images, before looks, or quick looks and are retained until the transaction completes.

The before images must be secured prior to the update of the data base. Writing of the before images to a retention file can occur with every record read or more efficiently only with reads for update. If the original data image is held in memory by the data base manager additional efficiency can be achieved by delaying the write of the before image until the update request actually occurs. This delay allows writing to the retention file only those before images that are actually modified. One further efficiency can be achieved by having both the unmodified before images and the updated record in memory at the same time. It is now possible to compare the original record with the updated record. This allows the before image contents to be reduced to reflect only the changed fields.

The other generally used method, referred to as deferred update, differs from the direct update method by delaying the actual data base updates until the transaction completes normally. The actual data base updates occur only when the transaction is committed. If the transaction aborts the data base has not been modified and requires no rollback. This method requires writing the updated records to a retention file prior to starting commit. These secured records are referred to as after looks and are identical to the after looks logged for data base restoration. This method has a window of vulnerability which exists from the point commit is started until all updates are successful. Since all updates are held until commit it is necessary to finish the process once started, that is roll forward with the updates. This window of vulnerability is referred to as commit in progress. During the commit in progress window the data base can be in an inconsistent state and rollback is not acceptable. This inability to rollback once in the window is due to the implied commitment to complete this transaction normally. In a transaction system there are additional actions which are dependent upon the synchronization with data base commit. An example could be sending a reply to the request concurrent with the commit of the database.

As with the direct update method, after look records can be reduced in size to the changed fields if the original record is still around when the updated record is returned to the data base manager. In both of the above methods improved performance is possible by reducing the number of writes or the amount of information being written to retention files. One example would be to write one reduced record to the retention file for both before image and after look. This record would reflect both the original contents and updated contents of the changed fields only. Since changes tend to affect minimum portions of data base records this could substantially reduce the amount of data retained. Combining the before images and after look records into one write reduces the number of writes while also providing capabilities for additional recovery solutions.

Data base restoration is the process of recreating the data base in an updated state after a catastrophic mass storage device failure. The first step of the process is to reload the data base from a previously archived version. The archived version is nothing more than a copy of the data base at a specified point in time. The copy could be either a static copy taken while the data base is not being accessed or a dynamic copy. A dynamic copy is taken while the data base is in service and is therefore changing as the copy is taken. The copy of the dynamic data base must retain both the starting and ending time to provide the interval of the copying window. The copy of the static data base need only maintain one time, either the start or end of the copy since the data base is not changing.

The information logged for data base restoration can be kept to a minimum and still allow restoration to a current state. In general, for the static archive case, the log needs to contain all the after looks and a record indicating those transactions which completed commit. The dynamic archive case requires the log also contain the before images if the direct update method is used for data base recovery. In all cases the log entries must carry an identifier specifying the transaction to which the entry is connected.

Restoring a data base proceeds by applying the logged records after the reload of the archived copy has completed. The steps will vary with the approach used to insure atomicity and whether the copy was static or dynamic. The simplest case is the static archive where it is necessary to establish those transactions which completed successfully. This can be accomplished by scanning the log from the time of the archive, accumulating a list of the transactions which completed commit. If the log is on magnetic tape this read can be backward to the time of the archive thereby positioning the tape for the forward reads. This list indicates the successfully completed transactions for which the after looks are to be written to the data base in a forward direction, from time of archive to the present. The logged after looks for transactions which were not completed are ignored and not applied to the data base.

Restoration from a dynamic archive follows the same steps as the static archive except it adds one more procedure if direct update is used. In this case it must apply, in reverse order by time, the before images for those transactions that were rolled back within the window of time during which the copy was occurring. This implies that the log must also contain before images and a rolled back transaction record. This rolled back transaction record will indicate those transactions which failed and were rolled back. Normally it is not necessary to retain this information for restoration. In this dynamic archive case the data base is being copied while the direct updates are occurring. It is therefore necessary to back out these dynamically captured changes. It also might be necessary to go backward in the log to a time prior to the start of the archive. Before images for transactions which were rolled back during the window may have been logged prior to the dynamic archive start.

The data base restoration process can be accomplished by a background utility. It should be followed by transaction system restart in order to rollback the effect of transactions active at the time of failure. This of course is predicated upon the retention file being available. The deferred update approach requires that active transactions which were in the commit in progress window be rolled forward, that is the commit completed. The direct update approach requires that the active transactions be rolled back. By doing this transaction system restart step it eliminates the need to apply before images to the restored data base. The exception being the dynamic archive window described above.

4. Distributed Transaction Integrity

The coordination necessary to provide atomicity and consistency across distributed transactions, which can comprise multiple diverse and dispersed local transactions, requires a means of synchronization. The methodology generally chosen is the protocol referred to as two phase commit. In an environment with multiple local transactions it is necessary to first insure that all the local units of work can complete normally and second to instruct these participants to commit. This two step completion process is the two phase commit protocol.

The distributed transaction environment may take on the topology of an inverted tree like structure with the top node being the initiator of the distributed transaction. In this topology the subordinate transaction systems are at the intersection of branches or the ends of the limbs. The players in a two phase commit scenario are the coordinator that starts the process by initiating the first phase and the participants that respond to the first and second phase. The coordinator is generally the top node of the transaction tree but this is not mandatory. The subordinate nodes of the transaction tree are the participants in the two phase commit process. The coordinator has the unique role of being the only node that can initiate the second phase, but only knows the existence of those participants immediately subordinate to itself. This means that middle level subordinate nodes take on the appearance of coordinator to those levels immediately below their level. Additionally at any level in the tree a participant knows only of the node above and those immediately subordinate to itself.

The first phase of the two phase commit process can be thought of as vote gathering in that the participating local transactions have the opportunity to affect the outcome. The first phase consists of the question "are you ready to commit". Another way of looking at the first phase of the process is that the coordinator is instructing the participants. The first phase then takes on the sense of a command "prepare to commit". Regardless of the sense of the first phase question or command the action required of the participants is to prepare to commit or rollback. That action is to secure all data images that are necessary to insure that the participant can either make permanent or back out the changes associated with this transaction. In the context of the above data base recovery methods this means that the before images for direct update and the after looks for deferred update are written to non volatile storage, both the retention file and the log when appropriate.

In the normal case there are two replys which participants may use to answer the first phase question. These are yes the data necessary to commit or rollback is secured, or abort this transaction. The general rule regarding a negative or abort response from any participant is that the distributed transaction will be aborted. This of course does not have to be the case, and distributed transaction networks can implement other rules. In those cases this would be an application dependent design decision. Those subordinates which answer in the affirmative are left sitting on the proverbial fence not knowing which way to fall, commit or abort, but prepared to fall either way. It is important to note that by responding yes the distributed transaction participant insures that it will commit if instructed to do so and may not later decide differently.

Conventions are established for the case where the subordinate transaction system has responded affirmatively but does not receive the second phase command. One of these conventions is referred to as presumed abort, which means if the subordinate does not receive the second phase command the transaction will be aborted. This of course is the safe action. However, there is the question as to whether presuming abort will be incorrect more times than not.

Additional responses, beyond the two mentioned above have also been defined. These responses are intended to reduce the need for the second phase in certain cases. In this role they might tend to improve performance by reducing the necessary communication over a network. One of these optional responses is read only which implies that no updates occurred therefore no further coordination is necessary. Another optional response is committed which implies that the participant has already committed the portion of the transaction for which it was accountable. This also implies no further coordination is necessary.

It has been pointed out that each participating node of a distributed transaction network needs the ability to recover the active local transactions. It is clear that when a local transaction aborts, the participating transaction system can independently rollback the local transaction while notifying the coordinator that it has aborted. This will generally cause the abort of the distributed transaction. However, it should not be presumed that aborting the active distributed transaction is the only choice. If the participating transaction system can recover the local transaction by rollback and restart from the input request using message recovery, then recovery should be attempted prior to notifying the coordinator. The notification to the coordinator with the results would follow the recovery attempt. This scenario can be taken one step further. If the transaction system is able to successfully restart and recover all or some active transactions then the system should come back online without aborting the recovered transactions. The coordination necessary to allow these scenarios requires that the participant first ask the status of the transactions in question. This is necessary since the coordinator may have already taken action to abort some or all of the transactions.

5. Considerations and Different Approaches

There are weaknesses with any recovery scenario regardless of the rules or conventions. Failures can occur where there is no expedient or practical means of recovery. The windows of vulnerability for these failures can be shifted or reduced in scope, but cannot be totally eliminated. This is also true of the two phase commit protocol which is useful in a heterogeneous environment where variations of the data base recovery methods are prevalent. However, it is naive to believe that two phase commit protocol will eliminate inconsistency across the network.

A case in point might be a distributed transaction incorporating multiple distributed nodes. The coordinator has issued the first phase question and received affirmative answers from all participants. The coordinator issues the second phase commit command but part of the network fails such that some of the nodes receive this second phase command while some do not. The participants that received the second phase command commit their data bases. The participants that do not receive the second phase command time out, and using the presumed abort protocol rollback their data bases. The network is inconsistent at least for some period of time. Furthermore, if the individual transaction systems dispose of the active transaction retention records upon transaction completion then this situation may not be correctable. If these temporary retention records are not released at transaction completion then the situation is recoverable with special effort.

The challenge is to reduce the period transactions are in windows of vulnerability, the frequency of the windows, or the magnitude of effort necessary to recover from a failure during these windows. Two phase commit reduces the period of time that the window is open across a distributed environment.

The direct update method of data base recovery breaks the updates into small windows at the possible expense of performance. This method essentially secures each update as the transaction progresses. This method insures that the data base is written at update and that the before images necessary for rollback are secured. The possible performance penalty occurs as a result of each update being a synchronous write. This is not a problem unless the system can take advantage of the potential performance benefits of the deferred update method. The direct update method provides another potentially important aspect for applications which wish to explicitly handle unrecoverable errors. All updating and securing is accomplished while the application is active allowing the application to take appropriate action in case of failures.

The deferred update approach reduces and moves the window of vulnerability by postponing all updates until transaction completion. This window is the previously discussed commit in progress window. If a failure occurs during multiple updates commit must still go forward. In this case the application program is no longer active in the transaction and cannot provide special unrecoverable error handling. The performance advantages of this method are achieved through both reduced and overlapped data base accesses. The updates can be overlapped on systems which provide multiple paths to the storage devices thereby reducing the total time to do multiple updates. Data base managers, which support main storage caching, can reduce the number of reads when a transaction attempts to access a record previously updated by this transaction.

Another approach to the distributed transactions commitment process would be to have all participants ready for either commit or rollback when they respond. This eliminates the need for the first phase and closes the windows created by the two phases. This can be achieved by using the direct update method consistently across all data base managers. This method provides the securing of before images and logging of after looks at the point when the data base is updated. There is no further processing except house keeping for this update when the transaction completes normally. The information necessary to rollback the update is secured in the event the transaction is aborted. In this case the coordinator knows that all participants are prepared to commit or rollback at any point that it has received successful responses from the participants. That is to say if the participants have not indicated an abort then it can be assumed that they are prepared to commit. When the coordinator enters distributed transaction completion processing, whether it be commit or abort, it simply issues the desired command.

A consideration for distributed transaction synchronization is the question of when the coordinating transaction system commits its local transactions. This question is not dependent upon the method of synchronization such as two phase commit. It is simply a question of the sequence involved when committing a distributed transaction across a network. Clearly there are three intentional choices in addition to the variation that might result from a communications network. First, the coordinator can commit those transactions and data bases local to its node before broadcasting the commit to the remote participants. In this case the coordinator might wait for the results, that is successful commit or unsuccessful, before notifying the remote participants of the direction on this transaction. This approach seems very questionable. If there is a weak link in the distributed transaction scenario it is most likely the network, implying that the second or third sequence would be more appropriate. Therefore, this sequence would not be recommended except in cases where a local transaction does not conform to the distributed transaction commit protocol as discussed below. In the second sequence the coordinator can broadcast the commit command to all participants with the individual commits essentially occurring concurrently. This sequence is the most egalitarian and presumes that each participating transaction system will complete the commits. This sequence is normally the method used for distributed transactions which can insure atomicity. The third sequence would send the commit commands to the remote transaction systems and wait for the confirmations before committing locally. This approach would be valid for cases where the coordinator knows there is only one remote and uses this knowledge instead of two phase commit. Another case would be for networks consisting of one node which does not conform to the two phase commit protocol. In this case the coordinator would send the first phase prepare to commit to those participants that supported the two phase protocol. If all answered affirmatively to the first phase then the nonconforming node would be instructed to commit. The confirmation from the nonconforming node would then control whether the second phase would commit or abort.

[illegible] of failures.

The deferred update [illegible] by [illegible] the [illegible] commit [illegible] the [illegible] performance [illegible] the [illegible] when a transaction [illegible].

[illegible] protocol [illegible] eliminates the [illegible] messages. [illegible] the coordinator [illegible] commit. When the [illegible] commit or [illegible].

A consideration [illegible] when the [illegible] network. [illegible] to the remote [illegible] the distributed [illegible] the [illegible] would be [illegible].

A Fault-tolerant Client-Server Transaction Model *

Paul Lockwood
Department of Computer Science
University of California
Santa Barbara, CA 93106

Divyakant Agrawal
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

We present a fault-tolerant client-server transaction model, which tolerates partitioning failures. The proposed mechanism is based on Gifford's quorum protocol for replicated data and, hence, has the same behavior in the presence of failures. We have implemented a prototype of the quorum based transaction model on top of Unix 4.3 BSD. The prototype supports concurrent execution of a procedure call by several clients and it guarantees failure atomicity of a procedure call. The results of some of the experiments conducted on this prototype are included in the paper. These results establish the viability of our approach.

1 Introduction

The *client-server* model has been proposed to provide services in a distributed environment [10]. The remote procedure call (RPC) has proven to be a very useful tool for implementing *client-server* interaction in a distributed environment. In this paper we propose a model for client-server interaction based upon an RPC mechanism that is tolerant of failures in a system. That is, client-server interactions remain available in spite of site and communication failures. The proposed scheme has been implemented on a network of SUN workstations.

The idea of RPC is quite elegant and simple in the absence of site and communication failures. By combining the technique that ensures *failure atomicity* of transactions [6, 4] with the RPC mechanism, we can achieve *exactly-once* semantics for the client-server transaction model [10]. Note that exactly-once semantics is the best that can be done for client-server communication in the presence of failures.

There is an added dimension of complexity which arises in a network when failures are taken into consideration. In a client-server model, we would like that a service be fault-tolerant and reliable, remaining available even when failures occur in the system. A fault-tolerant and highly available implementation of the client-server model can be provided by replicating a service (or a remote procedure) at several sites in the network. The price paid for achieving high-availability of services through replication is that it introduces complex synchronization problems of maintaining all replicas of a service consistent and up-to-date.

A large number of algorithms exist to maintain the consistency of replicated data. However, most of these algorithms are proposed in the context of databases [5]. There exist a few systems that implement fault-tolerant remote procedure call mechanism [3, 2, 13, 11]. However, most of these systems tolerate a restricted set of failures [3, 2, 13] and some of them have a very complex recovery and reconfiguration mechanism [11].

Gifford proposed the quorum protocol [5] for replicated databases that is resilient to site and communication failures even when such failures cause network partitions. One of the main advantage of the quorum protocol is that recovery from failures is very simple and straightforward. In fact, the protocol does not require that a recovering replica be made consistent or brought up-to-date. However, the problem with the quorum protocol is that it is inherently geared towards databases where data is delivered to the client's site and the execution takes place at that site itself. The quorum protocol needs to be embellished before it can be used to implement a fault-tolerant remote procedure call mechanism.

In this paper, we propose and implement a quorum based client-server model. A client in this scheme initiates a procedure call by sending its request to a quorum, which is a subset of the replicated servers. Each server that executes the call returns a version number, which is used by the client to verify that all servers in the quorum were up-to-date when they executed the call. This mechanism is optimistic in the sense that the client executes a call assuming that the servers are in a consistent state. A finite probability exists that the call may have to be retracted because the assumption does not hold. An update propagation mechanism is integrated to maintain all replicas

*This research is supported by the NSF under grant number CCR-8809387.

of the service up-to-date and to minimize the probability that a call will need to be retracted. Also, the propagation mechanism is used in combination with the version numbers in such a way that the re-execution of retracted calls would always be successful. Thus, in the worst case, only one extra round of message is needed to execute the remote procedure call successfully. The advantage of our scheme is that it is resilient to a larger class of failures, recovery from failures is simple and straightforward, and in some cases the proposed scheme is more robust to failures than other mechanisms that guarantee failure atomicity [6].

In Section 2, the model for the distributed system is defined. The update propagation mechanism and the protocol are presented in Section 3. In Section 4, we describe the design of a prototype implementation of the quorum based client-server mechanism on Unix 4.3 BSD and present preliminary results obtained from this prototype. We conclude the paper with a summary of our results in Section 5.

2 Model

A distributed system consists of a set of distinct sites that communicate with each other by sending messages over a communication network. There is no assumption about the underlying topology of the network nor is there any assumption about the existence of a broadcast facility. A broadcast or a multicast facility, however, can be exploited to improve the performance of the system. The only assumption that is made about the underlying communication network is that every site in the network has the capability to send a message to any other site in the network when there are no failures.

We assume that sites are either *fail-stop*, or may fail to send or receive messages. Communication links may fail to deliver messages. Combinations of such failures may lead to *partitioning failures*, where sites in a *partition* may communicate with each other, but no communication can occur between sites in different partitions. We also assume that the failures in the system are temporary. That is, a site does not fail permanently and the network does not remain partitioned forever.

There can be several servers resident at a site, and failure of any one of them does not necessarily affect other servers at that site. In this paper, we will be referring to server failures instead of site failures. A server implementation does not encompass multiple sites, *i.e.*, each server exists in its entirety only at one site. In order to increase the availability and, thereby, fault-tolerance of a service, we will replicate the service at several sites. A *server group*, SG, is a set containing all servers that are equivalent, *i.e.*, all servers with the same external specifications. Note that the implementation of each of these servers need not be identical as long as they have the same external behavior. SG may implement one or more remote procedures which are denoted as op. Also, the number of servers in SG is denoted by n. Since we are using the quorum protocol for replica synchronization, each procedure, op, has a number associated with it, denoted by q_{op}, referred to as the quorum size of the procedure op. The significance of q_{op} is that in order to execute a remote procedure call, op, it must be executed at at least q_{op} servers in SG. An operation's *server quorum*, SQ_{op}, is a subset of servers from SG such that $|SQ_{op}| \geq q_{op}$. Operations are classified into two categories, *query* operations that do not modify the state of the server and *update* operations that do modify the server's state. The quorum sizes of the operations are chosen such that certain quorum intersection properties hold. If op_i and op_j are two operations such that at least one of them is an update operation then:

$$q_{op_i} + q_{op_j} \geq n + 1 \quad (1)$$

Thus, the restrictions on quorum sizes of various procedures implemented by SG will depend upon their specifications. In this paper, we assume that the quorum restrictions are always specified for each remote procedure.

Finally, failure atomicity of a remote procedure call mechanism is guaranteed by using the notion of *atomic transactions* in distributed databases [6]. Furthermore, if SG interleaves operations of several clients, we assume that the servers implement a suitable concurrency control protocol, *eg.*, two-phase locking protocol [4, 1], to guarantee serializability.

3 The Client-Server Transaction Model

In this section, we present a quorum based model for fault-tolerant client-server transaction processing. Quorum based mechanisms present a problem in that all servers of a SG may not be in a consistent state. However, for remote execution of op_i to be successful our model requires that all the servers in SQ_{op_i} must be consistent. Therefore, a mechanism must be provided that guarantees that the states of servers in SG are consistent or that out-of-date servers within SQ_{op_i} are brought up-to-date so the op_i can be executed. This mechanism must be concerned with only those operations that change the state of servers, *i.e. update* operations. Our model includes an *update propagation mechanism* to minimize the chance that servers of SG may be inconsistent and to bring out-of-date servers in SQ_{op_i} up-to-date so that op_i be can executed. Next, we describe the update propagation mechanism used in our scheme, and then present the quorum based client-server model integrated with the propagation mechanism.

3.1 The Propagation Mechanism

A common technique to propagate information efficiently in a network and, thus, synchronize various components of a distributed application is to construct a *log* of certain application specific events that have occurred in the network. In the case of a server group, SG, consisting of replicated servers, such events include update operations executed at a server. Each server maintains a local copy of the log, which is organized as an ordered sequence of event records, and a propagation mechanism is employed to keep the copies of the log up-to-date. The mechanism makes use of communication operations, *send* and *receive*, to exchange portions of the copies of the log for this purpose [12, 8]. The background messages used in the propagation mechanism to bring all the copies of the log up-to-date are also referred to as gossip messages in [8]. We have chosen the algorithm proposed by Wuu and Bernstein [12] to integrate the propagation mechanism with our protocol.

Wuu and Bernstein [12] describe an efficient implementation of the propagation mechanism. Each server, s_i, maintains a time-table, T_i, which is an $n \times n$ array of timestamps of events that have occurred in SG, where n is the total number of servers in SG. A server uses the time-table to place a bound on how out-of-date other servers are about events that have happened in SG. The time-table enables a server to decide what portion of its copy of the log should be sent to another server, and when all servers in SG have learned about a particular event. The latter information is used by the server to determine when certain portions of its copy of the log can be discarded. Hence, a server retains a particular event record in its copy of the log only if it is not certain that all other servers have learned of that event. The *happened before relation*, "$\rightarrow$" [7], relates the application specific events and the communication operations employed by the propagation mechanism. Periodically a server sends its time-table and a portion of its copy of the log to another server. On receiving such a message, a server updates its copy of the log by including event records of which it was unaware and updates its time-table using information in the received time-table. The following two properties are guaranteed by the algorithm:

Propagation Property. Every server in SG eventually learns of each event in SG.

Causality Property. If e_1 and e_2 are two events such that $e_1 \rightarrow e_2$, then if a server knows of e_2, it must also know of e_1.

The propagation property is dependent on the assumption that server failures and network partitions are not permanent. It follows from the causality property that a server can process events in the *happened-before* order.

3.2 The Fault-tolerant Client-Server Transaction Execution

We now describe the execution of operations in the quorum based client-server transaction model. Each server s_i in a server group SG implements a set of operations, $\{op_1, op_2, \ldots, op_N\}$. Clients use these operations to access and update the copy of the data-object maintained by s_i. The quorum rules enforce the requirement that a client perform an operation op_I on q_{op_I} servers. A version number is associated with each copy of the data-object at s_i, and is initialized to 0. A *query* operation, which does not modify the state of the data-object, leaves the version number unchanged. On the other hand, an *update* operation, which changes the state of the data-object, increments the version number by one. Furthermore, the return parameters of an operation or a remote procedure call have a version number associated with them. The return parameters of a query operation at s_i are assigned the current value of the version number of the copy at s_i, and that of an update operation are assigned the incremented value of the version number. In addition to a copy of the object, each server maintains a copy of the log that includes operations that need to be propagated to other servers in SG.

The sequence of steps illustrated in Figure 1 is termed as the *normal mode* for execution of a transaction when all version numbers returned by the servers in SQ_{op} are the same. The case when the version numbers are not the same is termed as the *anomalous mode* of execution and is described in Figure 2. The commitment of the calls executed are carried out at the client and at the respective servers in the quorum as illustrated in Figure 3. The commit at the server results in the resources such as locks held by the client being released. Also, if op_l is an update operation then an operation record is appended to the copy of the log at s_i. This operation is later propagated to other servers, which are not in SQ_{op_l}, in the server group.

In order for the normal mode of execution to occur most of the time, an update operation must be propagated from the servers in its quorum to the remaining servers in the server group. This is accomplished by exchanging the copy of the log and time-table among the servers periodically, and it results in all copies of the data-object becoming up-to-date in a server group. When a server s_i receives a propagation message, it discards all operations in the message with version numbers smaller than or equal to vn_i. The rest of the operations are applied to the copy of the data-object in the increasing order of version numbers associated with the operations. Finally, s_i discards all operations from its copy of the log that have been communicated to all other servers in the group.

Note that since all servers in a quorum insert an update operation in their respective copies of the log, duplicate operation records may exist in the log. However, since

Client T	Server $S \in SQ_{op} \subseteq SG$
send(*op*, *params*, $\perp$) to all $S \in SQ_{op}$	
	receive(*op*, *params*, $\perp$)
	executes *op* yielding *results*, vn_i
	reply(*results*, vn_i) to T
receive(*results*, vn_i) from all $S \in SQ_{op}$	
checks that $\forall s_i, s_j \in SQ_{op}, vn_i = vn_j$	

Figure 1: Normal Mode of Execution

Client T	Server $S \in SQ_{op} \subseteq SG$
send(*op*, *params*, $\perp$) to all $S \in SQ_{op}$	
	receive(*op*, *params*, $\perp$)
	executes *op* yielding *results*, vn_i
	reply(*results*, vn_i) to T
receive(*results*, vn_i) from all $S \in SQ_{op}$	
$\exists s_i, s_j \in SQ_{op}$ such that $vn_i \neq vn_j$	
Computes $vn_{max} \geq vn_i \; \forall s_i \in SQ_{op}$	
send(*op*, *params*, vn_{max}) to all $S \in SQ_{op}$	
	receive(*op*, *params*, vn_{max})
	if $vn_i < vn_{max}$
	waits until ok(vn_i, vn_{max}, op)
	executes *op* yielding *results*, vn_i
	reply(*results*, vn_i) to T
	else
	propagate update to all $s_j \in SG$
$\forall s_i \in SQ_{op}$ such that $vn_i < vn_{max}$ do	
receive(*results*, vn_i)	

Figure 2: Anomalous Mode of Execution

Client T	Server $S \in SQ_{op} \subseteq SG$
send(*commit*, *op*) to all $S \in SQ_{op}$	
	receive(*commit*, *op*)
	commits *op*
	reply(*ack*, *commit*, *op*) to T
receive(*ack*, *commit*, *op*) from all $S \in SQ_{op}$	

Figure 3: Steps Taken to Commit a Transaction

version number uniquely determine an operation, they can be used to suppress duplicates. That is, only one operation record per version number need to be maintained. Also, this property can be exploited to expedite the process of discarding operation records from the copies of the log [9].

The case when some of the servers in SQ_{op} are not up-to-date when *op* is invoked is handled as illustrated in Figure 2. This is termed as ***anomalous mode of execution*** and an extra round of messages is required to complete the call. In this mode, the operation can be re-done when $vn_i = vn_{max}$ for *query* operations or when $vn_i = vn_{max} - 1$ for *update* operations. This is the condition for which the function *ok* returns true. Note that a list of out-of-date servers can also be sent by the client when it realizes that servers in SQ_{op} are inconsistent. This will ensure a rapid propagation of information from up-to-date servers to out-of-date servers within SQ_{op}. Also, the client can partition the list such that each up-to-date server sends propagation messages to an approximately disjoint set of out-of-date servers. Finally, we would like to point out that even with an extra round of message, needed in the anomalous mode of execution, a simple quorum protocol without a propagation mechanism is not enough to implement a quorum based client-server model. The formal argument of correctness of the mechanism presented in this paper appears elsewhere [9].

4 The Fault-tolerant Client-Server Implementation

In order to evaluate the performance and understand the behavior of the proposed model, we have implemented an application prototype that is based on this scheme. In this section, we describe the design and implementation of the prototype in the context of Unix 4.3 BSD, and present preliminary results of the experiments conducted on this prototype. The primary aim of the experiments is to determine if the quorum based client-server model is a viable paradigm for designing fault-tolerant transaction systems. The prototype is implemented on a network of Sun workstations (3/50 and 3/60) running Sun Microsystem's OS Version 4.0 of Unix 4.3 BSD operating system. The workstations are connected by a 10 Mb ethernet. The client and server modules are implemented as ordinary user programs in Unix 4.3 BSD, and the communication interface between these modules is based on datagram sockets of Unix 4.3 BSD.

4.1 Unix Implementation

The application chosen for the prototype implementation is the classical problem of ***distributed dictionary***. The reason for this choice is the wide spread use of the dictionary data type for implementing name servers, mail servers, *etc.* in distributed systems. The distributed dictionary service provides the following operations:

1. *lookup*(κ: *key*): returns the tuple associated with the key κ in the dictionary.
2. *insert*(κ: *key*; τ: *tuple*): inserts new key κ and the corresponding tuple τ in the dictionary.
3. *delete*(κ: *key*): deletes the tuple corresponding to κ.
4. *modify*(κ: *key*; τ: *tuple*): updates the tuple corresponding to κ with the new value τ.
5. *list*(): returns a list of key and tuple pairs in the dictionary.

Each server maintains a copy of the dictionary data-object and provides procedure call interface to execute the above operations.

There are three main components of the distributed dictionary implementation: ***client modules***, ***server modules***, and ***YP modules***. The YP module is similar to the yellow pages service in the Sun Network File System. This module is available at every site and it provides approximately up-to-date information about each server group SG, the network address of each server in SG, and the quorum sizes of each operation provided by SG. The server module is replicated at several sites and it uses the YP module to register itself and to determine the address of other servers in SG. Similarly, the clients use the YP module to locate the servers and the quorum size for the desired operations. The communication among these modules is accomplished through datagram sockets in Unix.

A server module consists of two processes: one to handle client requests, and the other to handle the propagation among other servers in SG. Since the interprocess communication in Unix is based on messages, we rejected the possibility of implementing the server module using two separate Unix processes. This decision was based on the observation that the overhead of message passing between the two processes in the server module will be prohibitive. Instead, we have implemented the server module using a single Unix process. The client requests and receipt of propagation messages are handled by the main body of this process. The dissemination of propagation messages is implemented by employing an interrupt handler. The interrupt handler is activated by a timer clock, which enables the server process to send propagation messages periodically to other servers. This solution is more efficient but is more complicated since interference between interrupt handler and the main body of the server must be prevented.

Each server maintains four data objects, a dictionary, an update log, a matrix clock and a recovery log. The dictionary comprises of student records and other related data fields associated with each record. The update log

pending list	redo list	waiting queue	student record
read lock	write lock	version number	log id

Figure 4: Fields for Each Data Object in the Dictionary

contains operations to be propagated to other servers in the *SG* and the matrix clock is used to determine when an operation can be discarded from the update log (*i.e.* this server knows that every server in the *SG* has learned of the operation). The recovery log allows the server to perform local recovery from site and server failures, restoring the server to the state prior to the failure. The recovery log is not being used currently, and will be employed in the future expansion of this prototype. The method for recovering from site and server failures is the *re-do* strategy, selected to maintain consistency with the protocol which requires re-doing operations in the anomalous mode of execution.

The dictionary is an array of records each of which has eight fields as shown in Figure 4. Student records contain the effects of only those operations that have been committed. For this reason, with each student record there is a *pending list* containing the operations executed but not yet committed. Each entry in the *pending list* contains information identifying the client that made this request, the operation and it parameters, and a copy of the student record that is the result of executing this operation, including the resulting version number. When a client requests that an operation be re-done, the operation is removed from the *pending list* and is placed in the *redo list*. Entries in the *redo list* contain information identifying the client, specifying the operation to be re-done and its parameters, and a version number indicating the version number for the student record before the request can be re-done. Requests to be re-done wait until an update propagation message brings the student record up-to-date so the request can execute using the current information. The *two phase locking protocol* is used to maintain consistency and to synchronize conflicting requests from different clients. Client requests that are blocked, waiting for a lock to be granted are put in the *waiting queue* until operations complete that allow the lock to be granted and the request to be executed. Each node in this list contains data that identifies the client, the operation the client requested and the operations parameters.

Only committed operations are stored in the dictionary to simplify the process of bring the server up-to-date, whenever update propagation messages are received from other servers in the *SG*. Any pending request for a student record for which an update operation has been received will have to be re-done, since it is obvious that the pending request was originally executed using out-of-date information. If the effects of this request were recorded in the dictionary the operation would have to be un-done before the update operation could be executed. The time to process an update message is reduced and the approach simplified by only recording the effects of committed operations in the dictionary. This approach also maintains consistency with the anomalous mode of execution and the future implementation of the strategy of recovering from failures, in each case operations are re-done.

The *update log* contains information that duplicates information stored in the *recovery log*. The cost of duplicating this information is justified by the fact that it would be too costly to search the *recovery log* every time an update message is sent to the other servers in the *SG*. This is critical in this implementation since the update message is formatted by an interrupt routine which must execute promptly so that the server may return to processing clients requests. Even if two processes were used in a uniprocessor machine the time efficiency of the update propagation mechanism is still of primary concern. Another factor that must be considered is that the size of the *update log* remains reasonable except when failures occur.

The fields, illustrated in Figure 5, for each entry in the *update log* are used in the following manner. The recording site's local clock is used to record the order that the receiving site processes update operations. The reason for this field is not obvious, but is necessary because the size of UDP messages in UNIX are bounded by the size of buffers maintained by the operating system, approximately 4 kbytes. Any message larger than 4k would have to be sent in multiple packets. In order for the *Causality Property* to hold, the propagation mechanism must guarantee that if any portion of an update message was received the entire update message is received. Since we are assuming unreliable messages and did not want to provide for atomic transmission of multiple packet messages we limited the size of our update message. This field is needed since all update operations may not fit in one message and we need to send the update operations maintaining the *happened-before* relationship. For updates sent form server S_i to S_j, note that recording site's local clock for each operation is not included in the update propagation message, it is used by S_i to determine which operations to include in the update propagation message. The update operation and it's parameters are needed by S_j to execute the operation locally if necessary. The server, S_j uses the executing site's

recording site's local clock	operation and parameters	client id
executing site's local clock	executing site's id	version number

Figure 5: Fields for Each Entry in the Update Log

id and local clock to determine if it has prior knowledge of this operation. The version number is used by S_j to determine if it has already processed a duplicate for this update operation. The client id is used by server S_j to check if a duplicate operation is currently pending. If a duplicate operation is pending S_j has not yet received and processed a commit request for this operation. In this case S_j can commit the operation at this time, since only committed operations are propagated.

At present the *update log* is maintained in memory and the *recovery log* is not used. The future enhancement of this prototype we intend to store the *update log* on stable storage and will start using the *recovery log* to recover from failures.

4.2 Performance

We decided to perform simple experiments in which a single client continuously performs one thousand dictionary operations. Each operation is executed as a *transaction* [4, 1], *i.e.*, the operation results in both the execution and commit phases. The operations were chosen randomly such that the selection of any operation type was equally probable. The quorum for each operation was also selected randomly such that each server could have been included in the quorum with equal probability. Since we used a single stream of random numbers for both operation and quorum selections, the operation type distribution was slightly different for the three experiments (1 server, 3 servers, and 5 servers).

The first experiment was conducted on a server group SG that had only one server. The following two scenarios were considered:

1. Both the client and the server at the same site.
2. The client at the remote site.

Since the experiments were conducted in a laboratory that is used by others, we performed five runs of each experiment to account for unpredictable network traffic. The means and standard deviations of the time taken by the execution and commit phases of a remote procedure call are illustrated in Table 1. The low standard deviation in the remote case is probably due to the fact that the time for remote communication masks local variations, resulting in reduced standard deviation as compared to the local calls. We would like to point out that no effort has been made to optimize the implementation. Hence, all results should be interpreted with the *local* case as the reference point. Also, in all the remaining experiments the client was always located at a site different than the servers.

The next experiment was carried out on SG of 3 servers with q_{op} size set to 2. In this case, the propagation rate was varied from 500 *msec* to 4 *sec*. The results are depicted in Tables 2 and 3. The surprising outcome of this experiment is that the percentage of calls that need to be re-executed due to the incorrect state of the server is very small (Table 3). Furthermore, the propagation rate does not influence the call re-execution significantly. However, the latency of the re-executed calls increase as the propagation is made slower. Note that the standard deviation in the case of latency of re-executed calls is very high. This might be due to the very low percentage of re-executed calls, coupled with a high mean value. The results, however, should be interpreted carefully because the sample size (30 re-executed calls) is too small for a statistically significant result (compare Table 3 and 5). The only reason to use slow propagation rate would be to control the message traffic. However, the propagation mechanism proposed in [12] transmits messages only when there is some activity in the system. This observation and the results in Table 3 suggest that a small value of propagation rate should be chosen to control the latency of re-executed calls. Another point we would like to make is that even with a single client we generated a very heavy load, *i.e.*, the client invoked operations without any intermediate delay. The final experiment for the case of 5 servers also yields similar conclusions (Tables 4 and 5).

The prototype can handle concurrent calls from multiple clients (by using two-phase locking) and aborted calls of a client (by providing undo mechanism). The retransmission of lost messages and the elimination of duplicate messages is handled at the application level, since we use UDP (unreliable datagram protocol) for sending and receiving messages. Our future task is to evaluate the availability of a service, which is implemented using the fault-tolerant client-server transaction model in the presence of failures. Also, this extension will require using stable storage which is not incorporated at present with the prototype.

Recall that we rejected the two process organization of the server module due to the restriction in Unix that processes cannot share a single virtual address space. The current implementation, which is based on one process and an interrupt handler, is complex and cumbersome. The

Client's	Execution Time/Call		Commit Time/Call		Response Time	
Location	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
Local	6.5 *msec*	0.6 *msec*	4.5 *msec*	0.5 *msec*	11.3 *msec*	0.9 *msec*
Remote	10.2 *msec*	0.3 *msec*	6.5 *msec*	0.2 *msec*	17.0 *msec*	0.5 *msec*

Table 1: SG consisting of 1 server; $q_{op} = 1$

Propagation	Execution Time/Call		Commit Time/Call		Response Time	
Rate	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
0.5 *sec*	14.7 *msec*	0.6 *msec*	8.4 *msec*	0.4 *msec*	23.5 *msec*	0.6 *msec*
1.0 *sec*	14.9 *msec*	0.7 *msec*	8.2 *msec*	0.5 *msec*	23.8 *msec*	1.0 *msec*
2.0 *sec*	14.8 *msec*	0.7 *msec*	8.2 *msec*	0.2 *msec*	29.4 *msec*	1.9 *msec*
4.0 *sec*	14.9 *msec*	0.4 *msec*	8.3 *msec*	0.2 *msec*	52.9 *msec*	1.9 *msec*

Table 2: SG consisting of 3 servers; $q_{op} = 2$

Propagation	Percentage of Calls		Re-execution Time *per*	
Rate	Re-executed		Re-executed Call	
	Mean	Std. Dev.	Mean	Std. Dev.
0.5 *sec*	0.7%	0.1%	15.3 *msec*	2.9 *msec*
1.0 *sec*	1.8%	0.2%	18.4 *msec*	0.8 *msec*
2.0 *sec*	3.0%	0.2%	200 *msec*	60 *msec*
4.0 *sec*	3.2%	0.3%	900 *msec*	120 *msec*

Table 3: Re-execution information for the case of 3 servers

Propagation	Execution Time/Call		Commit Time/Call		Response Time	
Rate	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
0.5 *sec*	22.4 *msec*	3.9 *msec*	16.0 *msec*	1.7 *msec*	38.7 *msec*	4.0 *msec*
1.0 *sec*	21.5 *msec*	2.0 *msec*	16.1 *msec*	1.4 *msec*	38.2 *msec*	2.3 *msec*
2.0 *sec*	21.4 *msec*	2.5 *msec*	16.1 *msec*	2.7 *msec*	38.7 *msec*	3.0 *msec*
4.0 *sec*	21.8 *msec*	2.7 *msec*	25.7 *msec*	4.7 *msec*	55.3 *msec*	6.4 *msec*

Table 4: SG consisting of 5 servers; $q_{op} = 3$

Propagation	Percentage of Calls		Re-execution Time *per*	
Rate	Re-executed		Re-executed Call	
	Mean	Std. Dev.	Mean	Std. Dev.
0.5 *sec*	0.5%	0.1%	19.8 *msec*	3.9 *msec*
1.0 *sec*	1.1%	0.1%	24.6 *msec*	1.2 *msec*
2.0 *sec*	2.0%	0.5%	31.3 *msec*	3.1 *msec*
4.0 *sec*	2.8%	0.2%	266 *msec*	96 *msec*

Table 5: Re-execution information for the case of 5 servers

operating system Mach, on the other hand, permits shared memory and has the notion of threads. We intend to port our implementation to the Mach environment, which would simplify the design of the server significantly. The simultaneous requests from multiple clients can be executed as multiple threads within a server task. This would increase concurrency if the servers are resident on a multiprocessor.

Another drawback of Unix is that it only provides unicast communication which is suitable in a point-to-point network. A multicast facility, however, will be more useful in a broadcast network. Most operating systems such as Unix and Mach do not provide multicast communication. Since fault-tolerant systems involve replication, which often requires simultaneous communication with several sites, availability of multicast communication is necessary for building high performance fault-tolerant systems.

5 Conclusions

In this paper, we have developed and implemented a paradigm for designing fault-tolerant distributed systems. The quorum RPC mechanism is based on the well known quorum protocol for maintaining replicated databases [5]. The advantage of this mechanism is that it is resilient to a larger class of failures, recovery from failures is simple and straightforward, and in some cases the proposed scheme is more robust to failures than other mechanisms that guarantee failure atomicity [6]. In addition to proposing the mechanism, we have also reported the preliminary results obtained from the prototype implementation of the quorum RPC mechanism. The results clearly establish the viability of the protocol as a useful paradigm for designing fault-tolerant distributed systems. The performance can be further improved with multicast communication and thread oriented virtual address space.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[2] K. P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[3] E. Cooper. Replicated procedure call. In *Proceedings of the third ACM Symposium on Principles of Distributed Computing*, pages 220–232, August 1984.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[5] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[6] J. N. Gray. *Operating Systems: An Advanced Course*, chapter Notes on Database Systems, pages 393–481. Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] B. Liskov and R. Ladin. Highly available services in distributed systems. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39, August 1986.

[9] P. Lockwood. A Fault-tolerant Remote Procedure Call Mechanism: Design & Implementation. 1989. Masters Thesis, Department of Computer Science, University of California at Santa Barbara, In Preparation.

[10] B. J. Nelson. *Remote Procedure Call.* Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981.

[11] B. Oki and B. Liskov. Viewstamped replication: a new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 8–17, August 1988.

[12] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third ACM Symposium on Prinicipіes of Distributed Computing*, pages 233–242, August 1984.

[13] K. S. Yap, P. Jalote, and S. Tripathi. Fault tolerant remote procedure call. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 48–54, June 1988.

UNIX Extensions For High-Performance Transaction Processing

Larry Clay, George Copeland, Mike Franklin
MCC
3500 W. Balcones Ctr. Dr.
Austin, TX 78759

Abstract

The goal of the Bubba database machine project at MCC is high-performance transaction processing for complex applications. To accomplish this goal, Bubba depends upon a number of novel techniques including parallelism, a powerful interface language, and a streamlined software architecture. The cornerstone of the Bubba software architecture is the BOS operating system. BOS has been built by extending UNIX[1] *System V to include functionality that has been traditionally provided by database systems. This paper describes these extensions and their implications for improved performance and simplicity of transaction processing systems.*

1 Introduction

The goal of the Bubba project at MCC is high-performance transaction processing (TP) for complex applications [Bo88,Sm89]. Bubba is a parallel, shared-nothing database machine. It consists of a large number of nodes, each containing processors, disk and memory where a memory can be accessed only by the processors within the same node. Nodes communicate using a message passing hypercube interconnect. The interface language to Bubba is the FAD database programming language [BBKV87, DKV88], which provides a superset of SQL functionality that supports complex objects and a general purpose programming capability. Bubba was designed to run on off-the-shelf hardware components (but with some special features including an MMU with a small page size, an uninterruptable power supply, etc).

The Bubba software is built on a custom operating system called BOS (Bubba Operating System) [CCF89]. The current version of BOS was implemented by modifying a version of UNIX System V from AT&T. Extensions were made in a number of areas:

Memory Management

- A shared persistent object space (using a single-level store).
- Two page sizes.
- Automatic two-phase locking.
- Copy-on-write workspace.
- Buffer management support.
- Support for full or partial database scans.

Process Management

- Lightweight processes (i.e., *threads*).

[1] UNIX is a trademark of AT&T Bell Laboratories.

- A fast implementation of semaphores which calls the operating system only when blocking is necessary.
- Message based task control that provides for local or remote activation and termination of processes and threads.

Messages

- A message interface that includes: multi-casting; large numbers of message queues per thread; short control messages embedded in the message header; and multi-page message bodies that are moved with MMU re-mapping techniques rather than copying.

While these extensions to UNIX were made to emulate the BOS environment, the memory management extensions make reasonable additions to the UNIX environment for the support of Transaction Processing. In fact, the Bubba software can currently be run as single-threaded UNIX processes that use the single-level store to support TP.

The remainder of the paper is structured as follows: Section 2 describes techniques that are used in Bubba to improve performance. Section 3 describes how UNIX has been extended to support these techniques. Section 4 presents a brief discussion on how these techniques are used by the database system runtime software. Section 5 presents conclusions.

2 Techniques for TP Performance Improvement

Because of the shared-nothing architecture of Bubba, each node functions in many ways as a single database system. Within each node, Bubba uses a number of novel techniques to improve performance over traditional system architectures. These techniques are described below.

2.1 Single Level Store

Complex applications typically are built using a programming language (e.g. COBOL) with an embedded database language (e.g. SQL). This approach has inherent performance limitations. A major performance problem in such systems is the need for data translation (i.e., copying, format conversion, and pointer conversion) between the programming language and database environments during run time. Bubba eliminates this data translation using a single-level store (or persistent object space) and uniform formats (where all data of a given type uses the same format). The persistent object space is mapped into the virtual address space of each process on a node that accesses the database (i.e. any Bubba process).

2.2 Two Page Sizes

Many systems use a single page size as the unit of memory allocation, locking and disk I/O. This causes a compromise which reduces performance. A large page (e.g. a disk track) causes reduced memory space utilization due to internal fragmentation. A large page often causes simple page locking to be inadequate. A large page also causes long page copy time. A small page (e.g. a disk sector) reduces disk arm utilization for large accesses. A small page size also requires large tables and cluster indexes. Most systems strike a compromise between these two extremes, which causes the system to suffer all of these problems to varying degrees. BOS reduces these performance problems by directly supporting two page sizes, a small memory page (e.g. 512 bytes) for locking, physical memory allocation, and disk writes, and a large disk block (e.g. 16K - 64K bytes) for disk reads.

2.3 Locking

Bubba uses two mechanisms for concurrency control: 1) automatic two-phase locking of data and 2) non-two-phase locking using semaphores for indexes. For concurrency control reasons, the persistent object space supports two types of pages: data pages and system pages. Data pages are those that are subject to automatic locking. When a memory location in a persistent data page is read (resp. written), the page is automatically read (resp. write) locked. Lock faults occur when a process that accesses a persistent data page does not have the proper lock (read or write) for the attempted access. Lock faults are detected by the memory-management unit (MMU) using the read and write protection bits. When a lock fault is detected, a lock on the data page is obtained if possible. Otherwise, the process requesting the lock is blocked until the lock becomes available. The advantages of this automatic locking mechanism are that 1) it uses standard hardware to efficiently support an often-used function, and 2) programs need not be aware of locking or page boundaries. The latter reason is especially important for supporting the more general-purpose (compared to SQL) programming environment of FAD. No automatic locking is done for system pages or for pages that are not part of the database. A semaphore mechanism can be used to protect data on these pages.

2.4 Copy-on-write

Because of the general-purpose programming capability of FAD, an undo log is difficult to implement. Bubba eliminates this problem by using the copy-on-write workspace feature of the persistent object space data pages. When a transaction writes a data-page, it gets a private copy of the page (if the only in-core copy of the page has not been propagated to disk, then the page is really copied, otherwise the clean in-core page is used without copying). When the transaction commits, the workspace page is made the current version of the page. The locking mechanism described above insures that there is at most one workspace copy of a particular page at any time.

2.5 Buffer Management

The buffer management support is designed to eliminate the "double buffering" problem that forces many TP systems to bypass operating system I/O services. Because of the shared persistent object space, persistent objects can be directly accessed in these buffers. When a page fault is detected, the entire disk block containing the page is brought into memory. The page replacement mechanism will page out any unused pages from the block if necessary. The default page replacement mechanism can be overriden via special system calls. These include allowing pages to be fixed in memory or to be prefetched for read-only scans.

3 UNIX Extensions to Support the Persistent Object Space

A prototype of Bubba has been implemented on a 40 node FLEX/32 multi-computer. Each Flex/32 node consists of a Motorola 68020 processor, a Motorola 68851 MMU, a Motorola 68881 FPU, 2 Mbytes of on-card SRAM, and a VME bus with 4 Mbytes of DRAM and a 180 Mbyte Winchester disk drive. Inter-node communication is supported via a 4.5 Mbyte common memory available to all processors. The vendor supplied version of UNIX for the Flex/32 was a port of AT&T UNIX System V Release 2.2 which was extended by MCC to emulate BOS and support the BOS persistent object space for UNIX processes.

The persistent object space is memory that is shared among Bubba processes, and exists independently of any particular process, i.e., changes to persistent memory survive from one process until another process changes the same object in the persistent object space. Further, the persistent object space is mapped to disk, and memory resident pages are kept in Safe RAM, thus avoiding the danger of data loss from a power failure. Bubba processes attach to the persistent object space when they are started via a special system call.

There are two types of pages in the workspace: *system pages,* and *data pages*. *System pages* fit the traditional model of shared memory, while *data pages* have some additional properties. For Bubba, *system pages* are used to hold indexes and control information, while data pages hold the database data. Persistent memory is allocated in terms of blocks (disk-track sized memory areas). A block can begin with zero or more *system pages*, with the rest of the block composed of *data pages*.

The rest of this section discusses the features of the Bubba persistent object space.

3.1 Properties of Data Persistent Pages

Data pages are shared similar to system pages. However, there are three primary differences:

1. The first property of data pages is that they have automatic *read-write* locking. When a process tries to read a page for the first time, if the page is not write locked, then the process is granted a read lock to the page. Otherwise, the process blocks until the conflicting lock is released. When a process tries to write to a page for the first time, it is granted the lock only if the page is unlocked or the locking process is the only process with a read lock on the page. Otherwise, the process blocks until the page is unlocked. Information for automatic locking is kept with the Persistent Page Tables, and enforced with the *write-protect* and *valid-page* features of the MMU. Once a process has a lock on a page, the MMU checks the lock on each reference without any software involvement.

2. The second property of data pages is that when a process is granted a write lock to a page, it gets a physical copy of the page if it is dirty. The modified page is put back in the public object space when a process completes its transaction and calls a special *commit* system call. This feature is called *differential page mapping (*aka *copy-on-write workspace)* and allows modified, but uncommitted data, to be easily discarded if a transaction is aborted (i.e. the transaction's process is killed without making the *commit* system call).

3. The third property of data pages is that newly created pages are write locked. Any attempt by other processes to access the pages before the creator commits causes the accessing process to block. If the process aborts without committing, the uncommitted physical pages are discarded and the new virtual space is considered unallocated.

3.2 Blocks Versus Pages

As mentioned above, virtual memory is allocated by blocks (which correspond in size, and are directly mapped to disk tracks) and otherwise dealt with in terms of pages. This provides two benefits. First, since the expensive part of a disk I/O is the seek time, transferring a full block in one operation is only marginally more expensive than transferring one page and much cheaper than making multiple requests for multiple pages. For example, disk controllers with track buffering can read a track without any wasted rotational latency, while reading a page can still involve an average latency of 1/2 a rotation. Second, since system pages and data pages can both exist within the same block, the programmer can collocate index information with the data it points to. These two features result in fewer disk I/O requests. To further improve disk I/O the TP systems programmer can specify that certain blocks, not necessarily contiguous in virtual memory, should be collocated on the disk. This reduces disk seek time by improving locality of disk references.

3.3 Commit / Abort Processing

To complete the two-phase locking protocol on data pages, transactions must use a special commit system call. This call releases all of the transactions's locks and moves any local copies (i.e. differential copies) of pages into the persistent space. If a transaction terminates (or aborts) for any reason without making the commit call, then its locks are released and any differential pages are discarded without affecting the persistent object space.

3.4 Special Persistent Memory System Calls

Database software that runs on conventional operating systems often suffers in performance because general-purpose operating system paging policies conflict with the atypical needs of database processing [Mos86]. While the persistent object space attempts to implement virtual memory in a way useful to TP, the TP systems programmer sometimes has access to knowledge not available to the operating system (for example, when a sequential scan or range search is being performed). This special knowledge can be used to get a boost in performance using several special system calls ("performance hooks") which alter the paging policy provided by UNIX.

These special calls allow the programmer to: fix pages in memory; mark blocks as cached; explicitly obtain or release automatic locks; and cause pages to be read from or flushed to the disk or discarded. Fixed pages are ignored by the page replacement policy, and remain fixed as long as the process that fixed them exists. Pages belonging to cached blocks are also ignored by the page replacement policy, but remain cached regardless of the status of the process that cached them. Fixing and caching are used to keep either temporarily or permanently hot pages in memory (e.g., index pages used often within a transaction or root index pages that are always hot). When the programmer knows that a group of pages needs to be locked (or read from disk, flushed to the disk, etc.), he/she can do this explicitly with one system call rather than on a page by page basis. This is important for full or partial scans.

3.5 Semaphores

In addition to the automatic locks mentioned in the memory management discussion, a generalized mutual exclusion mechanism is provided that is not tied to explicit virtual addresses. Thus, hot index information can be controlled with semaphores, which is more efficient than using two phase automatic locks.

Semaphores provide a single access mutual exclusion scheme based on user supplied semaphores. For example, the user supplies semaphore structures that may "live" in either the processes heap space or the persistent object space and be collocated with the objects they control. Semaphores are implemented by P( ) and V( ) macros. These macros run in user mode and only call the operating system when it is necessary to block or unblock a process.

A classic problem with using semaphores on top of an operating system, is that the OS will time-slice, or block for some other reason, a process while it holds a semaphore. The semaphore will then be tied up, possibly causing other processes to block, while the holding process is not running. We address this problem by allowing semaphores to be acquired in three states:

- If a process holds a semaphore in *strong critical section* mode, then UNIX will not schedule the processor to another process, even if the current process has to block on I/O. This mode, on single processors, reduces to setting a bit in the process's private memory that tells UNIX not to re-schedule the processor.
- If a process holds a semaphore in *weak critical section* mode, then the OS will switch to another process only if the process blocks (e.g., on I/O, waiting for a message, page faults, etc.).
- If a process holds a semaphore in *normal* mode, normal scheduling is in effect and a process might be rescheduled if a higher priority process is ready to run, the current process exceeds its time-slice, or the current process blocks.

3.6 Implementation Restrictions

As extensions to a general purpose operating system, the Bubba persistent object space features have the following restrictions:

- They require the presence of certain hardware (at least a 32 bit virtual address space, an MMU that supports 512 byte pages, and a Safe RAM).
- The 32 bit virtual address space limits the maximum size of a database on a single node to less than four gigabytes.
- Access to the persistent object space is currently unrestricted, and only one workspace is supported. A more general implementation might limit access to the workspace as a whole, provide several workspaces in the context of the UNIX filesystem namespace, etc.
- Access within the persistent object space is also unrestricted. Any process can change any persistent page. Further, while data persistent pages have some protection from "runaway" processes that make random changes then abort before committing, system persistent pages are not protected from such processes.
- The current implementation of semaphores allows user processes to communicate with the kernel via shared memory. This communication puts a certain amount of trust in the non-kernel code that might not be acceptable on a general purpose operating system. At minimum, this type of access would have to be restricted to "trusted" processes.

4 Utilizing the UNIX / TP Extensions

An important benefit that is gained by the combination of features provided by the persistent object space is the ability to provide direct access to objects from compiled code. Bubba object management makes a distinction between small objects and large objects. In Bubba, small objects are those objects that are less than a disk block (i.e., track) in size. *The structure of small objects is known to the FAD compiler*. That is, the code generated by the compiler can directly access the internals of small objects without calling object management functions. The single level store allows the same generated code to access objects without regard to whether they are persistent or not. Objects that are not currently in memory will be automatically brought into memory using UNIX's page faulting mechanism. The automatic locking of data pages insures that concurrent access will be handled correctly, and the copy-on-write workspace management insures that committed data is not overwritten by uncommitted data. The ability to directly access objects from the compiled code removes much of the expensive run-time interpretation of object structures that must be performed by systems that do not support uniformity of persistent and transient objects.

Objects that are larger than a disk block can be stored using an index (either a B-tree or hashed) which maps subobjects into disk blocks. For example, a large set of tuples is stored with a B-tree that maps tuples into specific blocks based on the value of one of their attributes (e.g., a key). The tuples themselves are stored as non-indexed objects within their assigned block. Objects that can potentially be indexed are tagged to indicate whether they are non-indexed or indexed. When the compiled code detects that an object is indexed, it calls the Bubba index manager to access it. Since indexes are encapsulated within the index manager, the index manager is free to use various tricks (such as non-two-phase locking for indexes and prefetching of data blocks for scans) to optimize access to large objects. For both indexed and non-indexed objects, pointers are implemented using virtual addresses. Therefore, no translation of objects is necessary when reading from or writing to disk.

Figure 1 shows an example of an object whose index has two levels. The top level of the index (called the cluster index) maps sub-objects into blocks. The lowest level of the index (called the block index) is stored in the same block as the data that it indexes. Since data blocks are fairly large, the cluster index can typically fit in memory. Therefore, all of the subobjects of a particular object that fit in a block can be accessed with a single disk I/O. The indexes (shown in the shaded area of Figure 1) are kept on system pages that are not automatically locked upon access. These pages are protected using non-two-phase locking techniques. The data area is made up of small pages that are automatically locked when accessed. Note that the leaf blocks contain both system and data pages. This flexibility allows the lowest level of the index (which will be the largest due to fan out) to be clustered in the same block as the data it references.

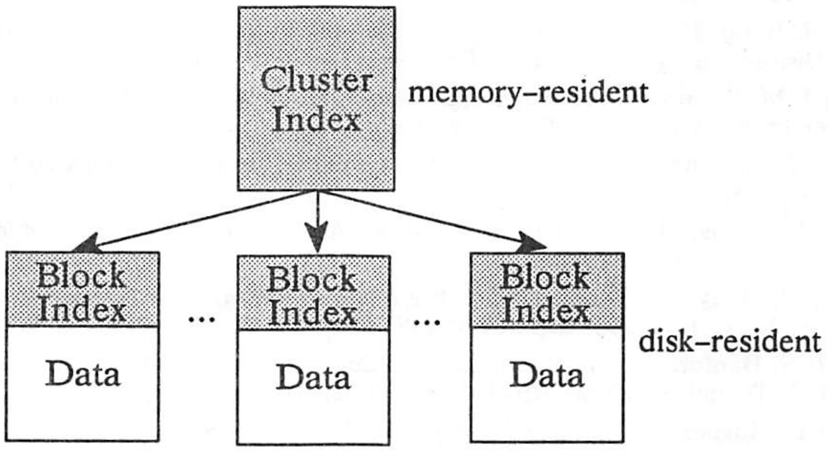


Figure 1 -An Indexed Object

The index mechanism's encapsulation of a structure that is sensitive to disk block boundaries supports efficient disk I/O. However, the direct access by the compiled code to the actual data within the leaf blocks allows efficient access to objects in memory. We believe that this combination represents a reasonable trade-off between disk access *and* in-memory efficiency.

5 Conclusions

While the Bubba software uses the BOS version of lightweight threads, messages, and message-based process control, any reasonable addition of these features to UNIX would suffice [BW88, Che88, Hug88, Tev87a, Tev87b, and Tha87]. Many of the single-level store extensions discussed in this paper are similar to proposals currently under consideration by UNIX International and the Open Software Foundation. Further, several vendors (including IBM and SUN) currently offer shared memory features which are similar to the Bubba single-level store. However, Bubba has a unique combination of features including automatic locking, the ability to commit the copy-on-write differential pages, and the use of two page sizes. Note that the IBM 801 CPR operating system [CM88] supports automatic locking and commit, but does not allow the mixing of system and data pages within the same block and depends on a custom MMU chip.

The techniques that we have developed in the construction of Bubba, will be used as the basis of a more general systems building platform, called Cworld, which is based on the C language and the UNIX environment. The design of this platform is currently under way.

Acknowledgments

There were many contributors to this document and the ideas it contains. Many of the ideas are due to Bill Alexander, Marc Smith, and several others in the Bubba project. Help in reviewing the document was given by Jim Browning of NCR, Herb Schwetman of MCC, and Ned Nowotny of MCC.

References

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, P. Valduriez, *FAD, a Powerful and Simple Database Language*, VLDB Conf., 1987.

[Bo88] H. Boral, *Parallelism in Bubba*, Invited paper, International Symposium on Databases in Parallel and Distributed Systems, Austin TX, 1988.

[BW88] J. M. Barton & J. C. Wagner, *Beyond Threads: Resource Sharing in UNIX,* USENIX Conference Proceedings Winter 1988, Februray 1988.

[Che88], D. R. Cheriton, *The V Distributed System,* Communications ACM, vol. 31(3), pp. 314-333, March 1988.

[CM88] A. Chang, M.F. Mergen, *801 Storage: Architecture and Programming,* ACM TOCS Vol.6 No.1, 1988.

[CCF89], L. Clay, G. Copeland, M. Franklin, *Operating System Support for an Advanced Database System,* MCC Technical Report ACA-ST-140-89, March 1989.

[DKV87] S. Danforth, S. Khoshafian, P. Valduriez, *FAD, A Database Programming Lanugage, Revision 2,* MCC Technical Report DB-151-85 Rev.2, Austin, 1987.

[Hug88] L. Hughes, *A Multicast Interface for UNIX 4.3,* Software Practice and Experience, vol. 18(1), pp. 15-27, January 1988.

[Mos86] J.E.B. Moss, *Getting the Operating System Out of the Way,* IEEE Database Engineering, September 1986.

[Sm89] M. Smith, W. Alexander, H. Boral, G. Copeland, T. Keller, H. Schwetman, C. Young, *An Experiment on Response Time Scalability in Bubba*, Proc. International Workshop on Database Machines, Deauville, France, June 1989,(also MCC Technical Report ACA-ST-379-88).

[Tev87a] A. Tevanian, R. F. Rashid, M. W. Young, D. B. Golub, D. L. Black, & E. Cooper, *Mach Threads and the UNIX Kernel: The Battle for Control,* Proceedings of Summer 1987 USENIX Technical Conference and Exhibition, June 1987.

[Tev87b] A. Tevanian, R. F. Rashid, *MACH: A Basis for Future UNIX Development,* CMU MACH Release 1, June 1987.

[Tha87] C. P. Thacker, L. C. Stewart, *Firefly: a Multiprocessor Workstation,* Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), ISBN 0-8186-0805-6, October 1987.

The Nested Top-level Lazy Server-based Transaction

Jeffrey L. Eppinger
Stanford University
eppinger@cs.stanford.edu

Transaction systems have existed for many years, but these systems only supported large, monolithic applications. New transaction systems, such as Carnegie Mellon's Camelot and IBM's Quicksilver, are being developed to provide inexpensive transactions in a general purpose environment. Key in the development of these general purpose transaction systems is the transaction model. Transaction models that do not support efficient operation logging will not provide adequate performance for accessing hot spots.

In this talk, I present an overview of the general purpose transaction model and describe the primitives that are frequently made available to programmers. I then present the problem of accessing hot spots and describe a general extension to the transaction model that allows hot spots to be accessed efficiently. This extension is the nested top-level lazy server-based transaction.

Nested transactions allow transaction writers to delineate sub-units of work. The nested transaction model provides two major advantages over the single-level transaction model: a nested transaction fails independently of its parent allowing the parent to take alternative action; concurrently running nested transactions are protected from one another allowing the parent transaction to safely execute sub-units of work in parallel. Once a nested transaction commits, its locks are anti-inherited by its parent. In other words upon commit, the nested transaction's locks become its parent's locks. The effects of a committed nested transaction are not externalized until its top-level ancestor commits. If an ancestor fails, the nested transaction's work is undone.

A nested top-level transaction commits independently of its parent. It is "nested" in two senses: the parent transaction waits for the nested top-level transaction to complete before continuing; if the nested top-level transaction aborts, the parent can take alternative action. The major motivation for using nested top-level transactions is that the locks obtained by the nested top-level transaction are dropped when it commits.

The controlled early release of locks is the property which makes nested top-level transaction desirable for accessing hot spots. Consider the hot spot in the debit-credit transaction benchmark: the history file index. Every transaction must obtain a write lock on the history file index, increment it, and then use the index to write a record into the history file. Careful design allows the history file index to be incremented by a nested top-level transaction; in the case the nested top-level transaction commits but the parent debit-credit transaction aborts, there may be a null record in the history file. In all cases, the nested top-level transaction reduces the duration of the write lock on this history file index to the duration of the nested top-level transaction. The latency of the nested top-level transaction is the key question. The cost of initiating and committing a transaction is very expensive relative to the cost of incrementing an index that is already in physical memory.

The server-based technique is used to reduce the cost of initiating and committing transactions. If a transaction is totally contained in one server process is it called server-based. The cost of initiating and comitting a server-based transaction is less than that of general transactions because no communication needs to be done. The server can construct its own transaction identifier and can commit the transaction unilaterally.

The lazy commit technique further reduces the cost of committing a transaction. When a transaction is committed lazily, the commit record is spooled to the log rather than forced. This allows transactions to get failure atomicity, but not necessarily permanence. This technique only makes sense when all the transaction's participants use the same log. A server-based transaction has only one participant and can be committed lazily. The lazy commit reduces the major latency cost of committing a transaction: the cost of the log force.

The nested top-level lazy server-based transaction is well suited for accessing hot spots. Consider again the increment of the history file index. The duration of the write lock on the index is is very short. After the lock is obtained, the index is incremented, the log records are spooled, and the write lock is dropped. Once the enclosing parent transaction commits, the log will be forced, guaranteeing permanence of the nested (top-level lazy server-based) increment transaction. Should the enclosing transaction abort, but the nested increment transaction commit, a null record will appear in the history file. If the nested increment transaction aborts, the parent debit-credit transaction will detect this and also abort (or retry). Of course, if both transaction abort, everything will be undone.

Nested top-level lazy server-based transactions have been implemented on the Camelot Transaction Processing Facility at developed at Carnegie Mellon University. These new transactions are used in the debit-credit benchmark as well as for transactional disk allocation and transactional dynamic storage allocation. Futher work needs to be done to generalize this technique for operation logging. This work has been done in collaboration with Dean Thompson, David Detlefs, and Randy Pausch.